The Mechanical Universe
An Integrated View of a Large Scale Animation Project

James F. Blinn
Jet Propulsion Laboratory
MS 510-110
4800 Oak Grove Drive
Pasadena, CA 91109

May 28, 1987

COURSE #6

Contents

Part I

OVERVIEW

Chapter 1

OVERVIEW

1.1 Introduction

The Mechanical Universe project required the production of over 550 different animated scenes, totalling about 7 1/2 hours of screen time. The project required the use of a wide range of techniques, motivated the development of several different software packages. This report is a documentation of many aspects of the project, ranging from artistic/design issues, scientific simulations, software engineering, and video engineering.

My interest in MU is twofold. One, to produce the material and two to see what tools need to be developed. It is real hard to develop tools if you dont know what they are supposed to do. Having a large animation project gives a lot of experience on what the problems really are instead of what somebody thinks they might be. This is a somewhat empirical approach to systems design. That is, several special case systems are built motivated just by the needs of some particular project. They are then analyzed to see what things they seem to have in common. In doing this sort of examination it is important to realize that you cannot prove that your assertions are correct in the same sense you can prove a mathematical theorem. The best that can be said is that the mechanisms described here seem to work well for the problems to which they have been applied.

1.2 What you should get from this course

Why am I doing this? My techniques may not be to the taste of everyone. But if you see something useful here, go ahead and use it. If you think the way this system works is real screwy, you are free to do it your own way.

The hope is that, out of all the experience in doing this project, there are some ideas that may be of use to other people building animation systems. I currently have no plans to distribute any of the software developed for this project. I feel my main product is ideas rather than hard code.

In these notes, I will tell about what I decided I should have done in addition to what I actually did.

1.3 Level of detail

There is always a problem, in describing a system, in determining the level of detail to go in to. When writing up something there is a tendency to think that many details are very attached to the specific system and are silly to bring up. It's a little like writing papers on how I organize my kitchen, which drawers have the silverware and which have the potholders.

But when reading about another system I find myself hungering for more detail to make the discussion concrete. It is comforting to know other people have to fiddle with same with same picky problems as I do. I will therefore go into perhaps more detail than necessary. It may seem at times to be almost a users manual for our particular system. but that's just to make things concrete.

1.4 Ways of breaking down the problem

Hey, I've got a great idea. Let's make the system *modular*! Well, we all know that modularity is good, the problem is what should the modules be? (diagram of circle broken in various ways)

Just making something modular doesn't mean it's good. Consider a system with a module for red pixels, another for blue pixels, etc. Or consider a program to "do anti-aliasing". These would be examples of bad modularity because the communication bandwidth into and out-of the modules is large. So, one criterion of good modularity is that it minimizes the amount of communication between modules. Other criteria may arise; thinking about this is still an ongoing process.

I have advantage in that, since I am the only user of the system I can experiment with different versions of it without a user community having problems. This is a disadvantage because the final system might be something that only makes sense to me and not to anybody else. I do have a *bit* of a user community, in that old scenes must continue to work, both for me and for people making slides.

1.5 Generality vs Special case

You can't properly see the general case until you have seen lots of special cases. Example: what's the next number in the list (3,5,7,...). Is it 9 (next odd number) or 11 (next prime)? This leads to the experimental approach to tool building.

Another philosophical issue: don't put stuff in unless it's really needed. This is to avoid the runaway features problem. If something is only used for one scene, try hard to do it another way. Only if several scenes would need some feature is it considered a good enough idea to put it in permanently.

1.6 Philosophy of exposition

These notes are organizes, as most books are, essentially as a tree, with sections, sub-sections, sub-subsection, etc. A particular system of knowledge isn't necessarily organized as a tree however, sometimes it's best thought of as a matrix. Some ideas are described here spread over several chapters, one on design, one on physical models, one on implementation. That is, things are grouped according to Science Concepts vs. Design Concepts; Mathematical Theory vs. Implementation of Theory. Some other ways of sorting things appearing here would be to take one idea and follow it thru from design to implementation in one chapter.

Part II

DESIGN

In this section I will discuss a few ideas on graphical design in general. The emphasis will be on concepts that are not specialized just to scientific animation, but may be applicable to other usages of visual communication.

I haven't learned this by any formal training. It has come by practice, intuition and perhaps genetics (I come from a family of artists). I learned to solve design problems by being presented with them, by being forced to think about the implications of color and shape choices. The results are what made sense to me at the time.

It is possible to come up with a few rules or guidelines to use in designing the images, but it's hard to remember to apply the rules all the time. It's not like programming. The concept of "works" or "doesn't work" is not so distinct. A program that doesn't work usually just stops dead and does nothing. A design that doesn't work just sits there and glares at you. The image always exists, it just might be confusing. There is more of a continuum of correctness in design.

Another question of interest to someone with a primarily technical background is: How much of design is just functionless decoration and how much of it is really necessary for making images easier to interpret? Some decoration really helps to understand an image. Other stuff just makes it look interesting and draws the viewer into the image. Watch for comments on this below.

Chapter 2

Graphical Design (static)

Static design refers to the appearance of a single frame. The concept of motion design is discussed in the next chapter.

2.1 What is a design Problem?

Let us begin with the question of "what is a design problem" anyway? It can be likened to pantomime. You are required to present some information that, perhaps, could be described in words, but you are required to use only pictures.

Some simple examples.

- The Voyager spacecraft approaches a planet. A moon is off to the side. You must pan across to see it but still give the viewer some idea of context of where he is now looking compared with where he was looking before.

- Look at the magnetic field of Uranus. The field is offset from the center by .3 times the radius of the planet. You want to show how this makes surface field have different strengths at the North and South magnetic poles. But close up you can't perceive the shape of the dipole field and its rotational motion, it's just a tangle of lines. You have to move in and out to show details vs./ global structure.

- How about a more detailed example. We will take an example from program 5, Vectors. The idea is to list the various types of vector expressions and to give an idea of whether the result is a vector or scalar. New items are added to the list as the program proceeds. The

whole list may not fit entirely on the screen. In addition, as a new item is added, some geometric demonstration is needed to show what it is.

Lets look at solution to this last problem. We represent an abstract "space" where the vectors live as a kind of vector land. There is a river running down the middle separating it from scalar-land. This allows us to display the lists in perspective receeding into the distance. As each new object is introduced it is added to the front of the list and list receeds further into the distance. Old list items may no longer be legible but the memory of them is enough to remind the viewer of what they are.

The key elements are:

Differentiate between Vectors and Scalars

Give an impression of 3D space but dont make it look too realistic.

An oblique view of the ground plane must appear to receed into the distane. This can be shown by texture. An obvious texture is a grid which shows perspective very well. However, at this point in the academic development, the notion of a coordinate system has not yet been presented. Some other textural effect must be used. Textrue mapping a random, say, pebbly texture would be slow. The resolution is to place a randomly scattered group of lines looking like grass across the plane. Just a few such lines can give a very cheap impression of receeding ground plane. Also the color of the plane is made to get bluer and paler as it moves into the distance.

Drop shadows help to bring out the 3 dimensional quality and make the vectors seem to hover above the plane, giving an interesting surreal effect.

Later in the program, when unit vectors and coordinates are introduced, the grid is placed on the plane (but only a small piece of it). Grids are a bit overused in computer graphics but for a lot of what we are doing in MU they are necessary since we are actually plotting graphs.

When we introduce unit vectors $\hat{\subset}$ and $\hat{\supset}$ they tip their hats. When showing the construction of a vector product, the term for *vector add* ane *vector multiply* are slid down close to the grid.

2.2 Direction of Attention

It is important to direct the attention of the viewer to the important parts of the picture. These scenes will be seen on TV in fairly brief bursts, so the important parts must stand out. One good trick for doing this is to look away from the screen and look back quickly; determine what you see first when looking back. Is that the important part of the picture? If not, change the picture to make it so.

This means avoiding gaudy backgrounds; the background should not look more interesting than the foreground. In one example I had an equation over a dark blue background that graded into orange, giving a sort of sunset effect. It was very pretty but the problem was that when you first look at the screen all you saw was the orange. I changed the background to a more neutral color and the first thing you now see is the equation.

2.3 Avoiding Information Overload

I have consciously avoided trying to "dazzle" the viewer. Dazzling implies an overload or numbing of the senses. The idea here is to communicate and draw the viewer in instead of making him tip backwards off his chair.

For the same reason I don't use lots of spinning or tumbling of 3D objects, it seems distracting to me. There is a tradeoff here between not giving viewers enough views of an object to be able to understand its three-dimensional shape vs. making it confusing by spinning it around too fast.

One important trick to encourage simplicity in design is for the designs to be done while viewing a monitor from across the room. If it can be made legible at a distance of 10 feet it's about right. This discourages putting in too much small detail.

2.4 Video Restrictions

You can't be subtle in video. You can't get a lot of detail into an image (for reasons described more fully in the Video chapter). Lines must be pretty thick; vertical and horizontal lines can come out different colors if you make them even as thin as the video bandwidth allows. Red comes out especially blurry. You can't have abrupt color changes, especially of complementary colors, (like green to magenta) across a scan line.

I regard this as an advantage, especially in designing moving things. It keeps pictures from getting too busy, although some of the scenes in MU are, even so, too cluttered.

Note that the slides shown during this talk were made at video resolution. Again, it forced me not to put too much on one slide. Many slides shown in talks have lots of microscopic detail that simply isn't legible. Viewing slides in an auditorium has about the same visual detail as video.

Don't design on a hi res system then transfer to video. You'll just fool yourself into thinking that some things will work, but they will turn to a blur on the transfer.

2.5 Color Selection

Given the color television medium we have both the opportunity to make scenes in color and the responsibility to make the colors look good. There are a few common tricks to use in color selection.

I find I have favorite colors, I tend to lean towards blues and greens. I don't like purple. I once used it purposely to break out of a rut, as a background in the scene on conic sections. I originally wanted to put a red cone in front of it but I couldn't get a red that didn't disappear into the purple in dark areas (as seen in b/w). Finally, I went to a brighter yellow cone.

2.5.1 Make it work in Black and White

When designing, look at the picture with the color turned off and see if it "reads" (to use a designer term). "Reads" in this context means "can you tell what is going on; do the appropriate things stand out"

While color is important in the MU animations it is not the only thing that differentiates items on the screen. It's not crucial. I have made consistent color decisions but the viewer is not expected to remember color schemes to understand a scene.

2.5.2 Context

Color selection programs are not too useful because colors always look different in context. The only real way to see how they look is to make an actual picture of the scene.

2.5.3 Distance Cues

Distance can be represented by making things disappear into the fog. This was done literally in a scene of the molecular arrangement of a salt crystal.

Other color cues: the color of things gets bluer and paler with distance.

Field lines are a complex set of three dimensional curves. they can look like a pile of spaghetti if you're not careful. The distance effect is aided by three things. One, normal depth cueing. Things get darker with distance. Two, drawing them in depth order so a closer (brighter) line will overlay a farther line. Three, making the intensity of the line darker at the edges than in the middle. This gives a slight "cylindrical" solid quality to the lines.

2.5.4 Not too many

Dont use too many colors.

There is a problem with running out of colors. There are more physical quantities to represent than there are easily distinguishable colros. You can't use saturation or value to distinguish things because sometimes these need to be adjusted depending on context, e.g., energy.

2.5.5 Consistancy

Make consistent usage of color schemes to recall previous results, as well as to differentiate diffrent things. We will discuss the color scheme extensively below.

Wasnt always consistent. Some problems described in ending section.

Paler colors for mass * something

Colored backgrounds for 2 integrations of grav law

Colored backgrounds for bringing external eqns to prove Keplers 3rd law

Blue texture for energy equation.

2.6 2D/3D Considerations

2D diagrams are easier to understand than 3D, especially when they are in motion. This is partly because labels keep getting in the way of 3D diagrams in some views. Most of the physics of the first term of MU is essentially 2D problems (like Keplerian orbits). These are kept 2D. The inherently 3D concepts are are torque and angular momentum. The punch line is, use 3D only when absolutely needed.

In fact, some 3D situations were simplified to 2D. For example, I used 2D for the Lennard-Jones atomic motion simulation and the ideal gas simulation. The actual physics is 3D, of course, but 2D shows the phenomena adequately and 3D would be really confusing.

In the second term were more inherently 3D problems. You must use 3D for electromagnetic fields. Many textbooks use 2D for fields but much is lost.

Also, 3D is used as a trick to put more text on screen. As the screen tilts back, more text fits. The top row might not be legible anymore, but we can remember what it was.

2.7 Making things stand out from background

Drop shadows help make things stand out from background. While they are good for labels on graphs, don't put a drop shadow on the plotted graph line because it detaches it from the grid.

Put 3D shadows for 3D vectors even if it is abstract shapes with no light source. Helps to see 3D shape by simultaneously giving two views of the object, a 3D view and a projection of that view on the xy plane. This is sort of what the Cubists were trying to do, show many views of an object at once. The shadow technique is more the way we are used to seeing and interpreting things.

Make background different value, pale colors.

2.8 Character Sets

I used the Hershey character set for all animated text. This might make purists choke but it's available and free.

I used a sans-serif type font at first. This was suggested by production people and I didn't know any better. I finally looked at math books and realized that there are typographical conventions in mathematics that had never floated to my conscious mind before. Scalars are in italics, vectors are boldface. When I did this on the screen the equations began to look like *mathematics.* Two or three of the first programs we did still have the old typeface but I went back and redid the animation for program 2 to use correct character sets.

In fact, for scenes where an equation is just supered over live action, and not animated, I made the equations with our system. The normal video producition character font generators simply can't make math look right.

General principle of character design elucidated by Tufte: make the letters look as different from each other as possible to ease reading. Note in gothic (Helvetica) they don't so much.

2.9 Realism vs. Abstraction

Images representing some real, physical object are often overlayed with labels, vectors etc. For such scenes, the real object is rendered with a simulated light source and shading (usually with a simple polygon rendering program). The mathematical abstractions are overlaid with a line drawing program (lines don't change thickness as they get closer or farther from viewer).

2.10 How its REALLY Done

A designer does not necessarily consciously go through a list of design rules when constructing an image. In fact, for the most part, I just LOOK at the picture as it develops and ask myself

> What don't I like about this picture? Hmm, this disappears into the background, that is too subtle.

Lots of design rules are, in fact, made up after the fact. This is true in many other disciplines also. Like the rules of music composition. Most composers just wrote down what sounded interesting to them, leaving it to the music historians to figure out what the patterns were.

Chapter 3

Graphical Design (dynamic)

From reading Thomas and Johnson's book you are left with the impression that animation is the highest form of human art. It encompasses all aspects of static art and adds timing and motion too. Motion design may well be the next great research topic in computer graphics. Any results shown here are very preliminary.

3.1 Stylistic Evolution

The global style of the animation changed over the course of the project. In the beginning the animation was more cute, at the end it was more abstract/informative. Some reasons: the beginning dealt with simpler physics, the ending had harder physics, it took all my energy just to get it across. Also in the beginning, more people had suggestions for cute things, by the end there were fewer outside suggestions.

3.2 Speed and Timing

Timing refers to how many frames it takes for a particular motion to occur. The animation system allows for any floating point frame number to be used as a keyframe, with interpolation between. But in the earlier programs I tended to use 10 frame time slots for a particular motion. This was simply because the frame numbers were printed out on the screen and they looked prettier if they were in units of 10 frames. This is about 1/3 second and

on retrospect it seemed to produce pretty fast, jerky motion. In later programs I began using 20 frame motions. This still makes things a bit predictablely clocklike but this is not all bad. The music composer for the series mentioned that all the animations seemed to have a rhythm to them.

Part of the problem with not being more creative with timing is that the previewing program usually ran about 6 frame updates per second (skipping over enough frames each time to make the global timing approximate real time). You could see the general sense of the motion but couldn't appreciate the subtlties of timing so well.

3.3 Interpolation

It is the popular wisdom in animation that spline interpolation is better than linear interpolation. It is smoother. Most of the animations were done with splined motion. However, later in the series I began experimenting with linear interpolation and found it quite pleasing. Let's face it, the algebraic motions represent mechanical operations' so why not make them mechanical looking? In this case non-natural (jerky) motion sometimes looks more interesting than smooth motion because it's (1) different and (2) contains more high frequencies at the key frames.

3.4 Incorporation of 'classic' techniques

There are various 'classic' techniques that are found in 'conventional' animation that apply here.

3.4.1 Squash/stretch

Squash and stretch refer to a distortion applied to the shape of an object when it undergoes acceleration. This is easily done by animating the x and y scale factor of an object. Before it begins to move it gathers itself up by shrinking in x, then it stretches out in x as it is moving, and when it stops it shrinks briefly and returns to its normal size. This wasn't done in MU as much as it should have been.

3.4.2 Overlapped motion

The concept of overlapped motion states that motion 2 should start before motion 1 is completed. This works well with character animation but I found it of limited usefulness in algebraic animation. In agebra there is just too much to follow as it is, without having the individual steps of a

derivation merge into each other. Making the steps disjoint in time gives the viewer a chance to absorb one step before another begins. I did make the x and y motion of an object overlap, but this just rounds off the corners of the motion.

3.5 Perceptions of speed

I found it interesing to discover how limited our perception of velocity is. Given two successive scenes, where an object moves, say 1.5 times as fast in the second scene, it is very hard to tell which is which. Since were showing velocity changes in a lot of the physics this was a problem. Most of the solutions involved representing velocity spatially as well as temporally by adding streaks or velocity vectors to moving objects.

Another interesting speed-perception discovery concerns double framing. One would think that all animation is ideally single framed. Double framing is just a cheapo economy measure if you don't have the computer time to do all the frames. Double framing looks jerkier. Buth there's another perceptual effect of double framing.

Double framed motion looks faster than single framed motion.

That is, if an object moves across the screen in 1 second, it will look like it is moving faster if it is animated as 15 frames double framed, than 30 frames single framed. This was alluded to in Thomas and Johnson's book on Disney animation. They said that motion was sometimes purposely double framed to give it a 'jaunty' look.

3.6 Character Animation

The highest form of animation is perhaps character animation. In it you try to convey thoughts and emotions with just two dimensional shapes. We had some need of character animation for MU but the system is not really designed for fluid inbetweening Disney style drawings.

Instead I adopted a style that goes back to my High School days. A friend of mine and I made some simple character animation with paper cutouts and hinged joints. The 2D shapes of the components were fixed, they could just rotate about joints. This technique is easy to simulate with our software system and comes off reasonably well on the screen. Characters don't continuously turn from left to right; they do it in just one frame. Knees dont bend during walk cycles. The feet of an animated soldier just scissor back and forth. The feet of the Greeks were made like a

pinwheel; four feet radiating from a center of rotation. Just the bottom two stick out from the bottom of the toga. Then just rotating the feet-object makes the walk cycle. This is not the fluid Disney style but it has its own primitive charm.

Part III

SCIENCE

Chapter Overview Part III describes the scientific aspects of the Mechanical Universe project. The subject matter can be grouped in two ways: according to the design principle used, and according to the physical concept presented. I will do a little of both, in two chapters. There may be some repetition between the chapters, but seeing ideas in different contexts is sometimes usefull too. The final chapter describes the mathematical and numerical models used to represent the physics.

Audience When doing something of this nature it is important to keep the audience in mind. I had a very specific audience in mind when designing these animations; myself, before I understood the concepts. For the most part, these are the explanations that I would have liked to have had, that would have made the most sense to me when I was learning physics.

Roots We are all products of our environment. I would like to mention some previous experiences that have affected my design notions here.

Lillian Lieber and Hugh Lieber are a mathematician/artist team that produced a series of charming books in the 40's. Hugh, the artist, had a very surreal sense of making mathematical symbology visually interesting.

Various Disney animations were produced for science and mathematics. Among these were "Man in Space" and "Donald Duck in Mathemagic Land".

George Gamow wrote several books popularizing physics. His best creation is the Mr. Thompkins series. In these books, Mr. Thompkins attends a physics lecture and falls asleep. In his dreams the physical point of the lecture is illustrated, usually by exaggerating the effects so that they were more noticable in daily life. Particularly memorable was a scene in the "Old Woodcarvers' Shop" where a sculptor makes atoms out of little green marbles (electrons) and little red marbles (protons).

The "Chem-studies" series of films, made for high school use in (date)? These had several, conventionally done, animations of molecular dynamics during chemical reactions. The motion of the atoms in these animations beautifully gives a sense of the energetics of atomic bonding. These were produced by David Ridgeway, who is on the national advisory committee to the Mechanical Universe.

The Bell Labs science films such as *The Unchained Goddess*, and *Our Mr. Sun*. These were directed by Frank Capra, a Caltech graduate, and also a member of the advisory committee to MU.

Finally, a telecourse from the past: Continental Classroom. This was for-credit course offered over television about 1960. It had classes in mathematics, physics and chemistry. When I was young I was interested in this

stuff but didn't know where to go for information. When I found this course I got up religiously each morning at 6AM to watch it. I understood only about half of it but it kept my interest in the subject alive. I hope that, with MU, I might be making a series that generates similar interest in a new generation of students.

Chapter 4

Visual Metaphors (Design)

In this chapter I will discuss visual metaphors for physics, grouped by design concepts.

4.1 Color

A normal textbook diagram has shapes, lines, text. In video we have, in addition, color and motion. The challange is how to use them. Motion usage is, for the most part, more obvious than color. Where there is some previous convention for color assignment I tried to use it. Where there was none, I had to invent one.

When referring to explicit color values, I will use the notation developed by Alvy Smith. Color is three numbers representing

1. Value or Brightness (0...1)
2. Hue going around the color wheel. Numerical quantities go from 0...6 for one cycle. 0=red, 1=yellow, 2=green, 3=cyan, 4=blue, 5=magenta.
3. Saturation. 0=neutral, 1=fully saturated.

A lot of different ideas were keyed to colors. Much of this was rather subtle and the animations never relied solely on the color to be understandable. I was left with the impression, however, that there simply aren't enough colors to have a unique one for *everything*.

4.1.1 For Dimensional Analysis

When physical abstractions such as accelleration or torque are represented in vector diagrams or algebraic labels they must be some color. Rather than just making all vectors and labels white I chose to institute a color scheme that is keyed to the units the quantity is measured in. These color schemes are maintained throughout the series. This provides for a sense of continuity and also gives the viewer a sense for dimensional analysis.

Also, I tried to avoid the temptation to get overly cute with the colors. Colors are used primarily for labels. Terms in equations are usually white, otherwise the equation tends to look like confetti. A term is shown in color only if the dimensions are important for a particular derivation.

Position, velocity, acceleration

Position, velocity and accelleration are the most commonly used quantities.

- position = green (1,1.8,1)
- velocity = yellow (.7,1.2,1)
- acceleration = red (1, .2,1)

There are several motivations for this general color scheme. As successive derivatives are taken, the color shows a smooth progression along the color wheel from green to red so there is a visual progression between the colors. (Actually, the reddening applies not so much to derivatives as to the division by time).

Acceleration is the most 'active' of the three concepts. But red means 'stop', not a very dynamic idea (although it takes decelleration to stop). This might be a counter argument for the use of this color. But red is also the most exciting, attention getting color. It shows that something is going on, and thus looks dynamic.

Green (as in grass) shows a static 'place like' effect.

This color scheme worked well when applied to a scene showing an abstract bycicle rider. The intent was to show elevation and slope. The normal color for informational traffic signs (green) was used to label the elevaion. The normal color for warning traffic signs (yellow) then labelled the slope.

Note that the colors chosen are not pure; the hue values are not integers. The exact hues were selected visually to look nice together. Exact primary colors tend to look boring.

Multiplying by Mass

Mass times accelleration gives Force. Mass times velocity gives Momentum. Force and Momentum were given the same colors as accelleration and velocity except that the saturation was reduced. I think of Mass as a sort of dark grey color, looking solid, like lead or iron. So adding grey to the above colors desaturates them.

Energy

Energy is a dark blue color. This was chosen to look sort of like a lightening bolt. Color is (.2,4,1)

Torque and Angular Momentum

Angular Momentum is a sort of rotational concept. I toyed with the idea of giving Angular momentum vectors a sort of barber-pole effect but it seemed too busy. Angular momentum is also mass times velocity times distance. Maybe a sort of pale yellowish green? But that would not make it distinguishable enough from the other two. Finally I decided to take off in a new direction and make it a pale blue.

Torque, the derivative of angular momentum, is of course lavender, blue with red added to it.

Area, Volume

These were made variants on the green color. Area is a slightly bluer shade. Volmue is a still bluer shade. Maybe I was getting too subtle here, but you have to pick *some* color, and it might as well be for *some* reason.

Actually this choice was not entirely conscious, and as a result the color for area is not exactly consistant over the whole series. For example, the color of Gaussian surfaces in the electricity programs was the position color, not the area color. This led to some problems when showing surface integrals. (expand on this) You do your best, but sometimes mistakes creep in.

4.1.2 Solid, Liquid, Gas

In the thermodynamics discussion there is a section on the states of matter. In particular a PVT diagram is separated into regions where a substance is a solid, liquid and a gas. These regions were colored as folows.

Solid - medium brown. An earth color, designates the solidity of ground.

Liquid - bluish. Like the color of water

Gas - white. A transparent color.

In the PVT diagram there is a region above the critical point where the distinction between liquid and gas disappears. VanderWaals' equation was used to find the degree of liquidity and used to calculate a saturation value smoothly grading from blue to white for this region.

4.1.3 Electric charge

Positive and negative charges are shown in many scenes. There has been a sort of convention for some time in engineering to make the positive leads colored red. In addition, the Gamow books represented electrons as green marbles. So a similar color scheme was chosen for the MU series. They are a somewhat different shade of red and green than acceleration and position.

But theres two problems here. First, not everyone has color television sets. So the colors were chosen so that, in black and white, they would still have enough difference in brightness to be distinguishable. Second, although red and green are complementary colors visually, in video it is red and cyan (a sort of pale blue). In some instances a neutral charge (e.g., for neutrons) is shown as, obviously, white. It would seem best to make the plus and minus colors add up to white. So a more bluish hue was chosen for negative charge. The exact value was actually changed during the second half of the series to be exactly cyan. This seemed necessary to make plus and minus add up to neutral but I'm not sure it was a good idea in retrospect.

4.1.4 Electric and Magnetic Fields

I've always thought of magnetic fields as blue, many published diagrams have shown it as blue. In fact, in an earlier project showing the magnetic field of Jupiter, I made the field lines blue. The question is, what color are *electric* fields? Since they are lines between positive (red) and negative (greenish blue) I decided to make it the color halfway between them, yellow. Note again that this is a different yellow than used for velocity.

4.1.5 Electic Potential, Magneic Potential, Flux

Electric potential, or voltage, is the line integral of electric field (yellow) dotted with displacement (green). Forgetting, for the moment, the displacement, we have here a similar situation to position being the integral of velocity. So a shade of green (moving in the non-red direction) was selected for voltage. Another way of putting it is that the electric field (yellow) is the derivative (more specifically the gradient) of the voltage (green).

Magnetic fields do not have an equivalent scalar potential, but if you integrate the magnetic field you get a sort-of potential. This was rendered a shade of blue similar to the field lines.

Flux is the integral of either of these fields over a surface. Flux (magnetic or electric) was represented as a paler version of the green and blue colors.

Again, we are starting to get overly academic and subtle here, but you do have to pick *some* color.

4.1.6 Temperature

Heat sources and heat sinks appeared in the section on thermodynamics. Typical hot and cold colors, red and blue, were used but the exact shade was adjusted to be different from colors already allocated for other purposes.

4.1.7 Invisible Light

Moving away from abstract concepts to more concrete phenomena, what about Infrared and Ultraviolet light? These needed to be represented in scenes depicting spectral lines and electron transitions. I originally tried using grey to mark the position of the line in the spectrum but it didn't look monochromatic enough. Spectral lines should emphasize their monochromaticity. You could outline the location of the appropriate spectral line with dotted lines but there really wasn't room, it would look too blurry. Also it might appear mysterious without some explanation. Finally used a dark red for IR and a dark violet for UV that fall in the "non-spectral" region of the human color space. That is, certain shades of purple have no spectral counterpart. They are only percieved if the eye recieves two frequencies (a red and a blue) simultaneously. These made an interesting, if paradoxical, choice for light that had no perceptable appearance, but came from a single frequency.

4.1.8 Frequencies

In the program on resonance there is a derivation which depends on the interaction of two frequencies of the system, the driving frequency and the resonant frequency. Two complementary, but fairly pale, colors were used to represent the frequencies both in the equations and in a geometric plot of the frequency summation. Here is a case where the geometry of a scene can be related to the algebraic representation of that geometry by color keying.

4.1.9 Relativity coordinate systems

There were many scenes in the relativity section that illustrated events as seen from two different reference frames. The two frames were usually those of a cartoon Albert Einstein and a cartoon Henry Lorentz. When we first see them, Albert is wearing a tan suit and Henry is wearing a blue suit. Thereafter, any algebraic or pictorial reference to Albert's frame is drawn in tan and any reference to Henry's frame is blue. These colors were initially selected as typical colors that suits come in, but they were fine tuned to show up distinctly in black and white and when placed on a common background. Actually, when I first decided to do this, I had make Henry's suit dark grey. But dark grey didn't look good as a comparison color to tan—tan and blue are more balanced complementary colors. I had to re-make one of the first animations just to change the color of Henry's suit. The production people probably thought I was nuts.

4.1.10 Wave/Particle Duality

The last three programs of the series begin to touch on quantum mechanics. Several of the scenes depicted wave-particle duality. A pair of complementary background colors were selected to represent particles and waves. All particle equations appeared over dark pale green, all wave equations and plots of wave functions appeared with a dark pale magenta background.

4.2 Literal vs. Schematic

My tendency is to be too literal. The sizes and timings of some phenomena sometimes have too big a range to make this easy. But since this is **Computer Animation** the viewer expects precision and accuracy.

When sizes or timings must be distorted into schematic diagrams, it is important to give some visual cues that this is being done.

One way to do this is to have the schematic scenes drawn with sketchy or irregular lines. This removes the precision effect of perfect lines.

4.2.1 Literal

Some things were done geometrically correctly, even though it was difficult.

For example, the radii of the orbits of the Bohr atom are proportional to the perfect squares (1, 4, 9, 16, ...). In order to see as many as four orbits, the scale must be too small to make the first orbit clear. This was usually solved by having the camera pull back when discussing the larger and larger orbits. This is a useful general principle, as it was described in

an earlier chapter concerning list receeding in perspective. If some things are too small, start close up and pull back.

4.2.2 Schematic

When force laws are introduced we needed to show the operation of gravitational and electric forces. At this point, the magnitudes weren't important, only the signs. Crude schematic faces were used as mass particles (grey faces) and as positive and negative charges (red and cyan faces). The motion was sketchy, only showing attraction vs./ repulsion, and the faces were sketchy, with irregular and comical lines. This visual signalling was not done enough in the series.

Other scenes with schematized motion included:

- A depiction of resistance in metals. The normal velocity of electrons in a metal is far greater than the drift velocity which is the electric current. Therefore an accurate depiction of current wouldn't look much different than random thermal motion. The relative velocities were made more equal for illustration purposes. Also, resistance is caused by collisions of electrons with imperfections and thermal motions of the atoms in the metal lattice. These are usually too few and far between to be easily noticable. They were made more obvious by flagging some metal atoms a different color and having the electrons bounce of them elastically, while not being affected by the positions of all of the non-flagged atoms.
- An electrical spark is generated by a chain reaction. Electrons are accellerated by an electric field and build up enough kinetic energy to knock other electrons off atoms. Again the typical spacing and oftenness of the real situation would not fit on the screen. Some exaggeration was done.

4.3 Recurring themes

There are several items that appeared in the series for many different purposes. These are described in this section.

4.3.1 Atoms

Any depiction of an atom is inherantly schematic, given the quantum nature of things. Some of the many representations of atoms were:

1. A three dimensional sphere.

2. A two dimensional disk.
3. A red blob representing the nucleus, surrounded by spinning electrons represented by rapidly rotating circular streaks.
4. A exponential blob representing an electron orbital smeared out in space.
5. Full 3D wave fuctions for the energy states of the Hydrogen atom. These were fuzzy blobs with the appropriate donut, dumbell etc./ shapes.

In some cases more than one representation is used in the same scene. For example, a scene depicting the crystal structure of Sodium Chloride appears in a program that contains a lot of spinning electron representation scenes. The crystal structure is represented by packing spheres together. To help people make the jump between the representations, some of the atoms are flown into the foreground and then made transparent, showing the spinning electrons inside.

4.3.2 The Teapot

I attempted to stick the teapot in every chance I got. This was usually used in scenes where some cannonical "arbitrary object" is needed. It appears in

- Program 4 - Inertia

  It is used as an example of things flying off the earth.

- Program 29 - General Relativity

  Einstein's though experiment concerned dropping objects in an accellerating rocket vs./ dropping things in a gravitational field. We needed some objects to drop. The objects were, and apple, a weight, and the teapot.

- Program 7 - Integration

  It is used a lot here. The program begins with the ancient Greeks calculating areas of simple shapes. They start out with a golden rectangle, the Greeks smile. Then they try curves like a circle or a parabola, the Greeks appear concerned. Then they are confronted with the teapot, the Greeks frown. Later, Kepler finds the volumes of sevral mathematical shapes but is dismayed when the teapot appears. When integration is invented and demonstrated on an abritrary functional curve, that curve is the cross secional outline of the teapot.

Finally, the volume of the teapot is found by shoving it through the derivative machine. It's 42. (actually I'm not sure what the actual volume of the teapot is in its original coordinate system).

4.3.3 Parchment

Many programs use the results of mathematicians from Kepler's time or before. These are shown as dark brown lines on a tan, mottled background, representing parchment. It is actually a simple 2D fractal pattern.

4.3.4 Shadowed Circuit Diagrams

Several programs depict the operation of simple electronic circuits. The dynamic operation of the circuits produced time varying voltages and currents. Current was represented as a marquee effect of dotted lines on the wires. The speed and direction of the marquee indicated the direction and magnitude of the current. The voltage was shown as a vertical skew distortion of the circuit diagram itself, the higher the voltage the higher the point (geometrically). To make this easier to see, the circuits were drawn floating above a dark grey ground plane. The shadow of the circuit on the ground plane didn't distort.

In addition, charge on a capacitor was shown as a red and blue glow on the plates and by showing the plates connected by yellow electric field lines. The magnetic field of an inductor was shown as a number of blue lines thru the inductor coils. The heat dissipated by a resistor was represented by a red glow around the zig-zag of the resistor symbol. The brightness or number of these colored indicators is proportional to the magnitude of the charge, field, or heat.

4.3.5 Molecular Dynamics

Many phenomena are described by the combined motions of a large group of atoms acting according to a simple force law. The most commonly used force law is derived from the Lennard-Jones potential. This generates a weak attractive force between atoms that are far apart and a strong repulsive force between atoms that are close together. When they just touch, the force is zero. The results of this force law can easily be simulated by a simple numerical integration on a collection of a few tens of atoms. The resulting motion is fascinating to watch.

The atoms are rendered as pale bluish disks at the appropriate position and radius. (Pale blue represents the negative electron cloud surrounding the atoms).

When molecular dynamics is used to illustrate heat flow, the velocity of the atoms is important. Subtle variations in speed are not easy to percieve so, in these cases, the velocities are represented by yellow velocity vectors superimposed on the atoms.

Chapter 5

Visual Metaphors (Physics)

Here are some more Visual Metaphors, this time grouped by subject matter, rather than by design issues.

5.1 Algebraic Ballet

To make the science respectable we had a lot of algebra to present. Algebra, hovever, can be a bit draggy. We decided to liven it up by animating the algebraic transformations that the equations go through. These animations usually go by fairly fast, in fact it is not likely that the viewer will be able to follow all the steps upon first viewing. The speed was a concern, but we felt that making it slower would slow down the programs too much. The idea is to get the feel for what is going on and be able to look at a videotape slower to get the detail later if desired.

Transforming algebraic operation into motion proved to be an interesting exercise. Many of the motions seemed pretty obvious to me, but they will be listed here for completeness.

5.1.1 Term labeling

It's easy to lose track of what different symbols in an equation represent. This was addresses by having the symbols identify themselves with english words popping out and shrinking back into them.

5.1.2 Balancing Act

Simple algebraic operations to move terms around were animated literally.

- Terms moving to opposite side of = sign. Adding on one side means subtracting on the other so a + or − sign flips it identity as the term hops over the =.
- Factors moving to opposite side of = sign. Multiplying on one side means dividing on the other. When a factor jumps over the = it lands below or above a division bar according to whether it came from above or below.
- Distribution. $a(b+c)$ becomes $ab+ac$ by having the a jump up, split in two, and each copy land next to the appropriate term.
- Squaring. Either two 2's come down from above and land on each side of the =, or a 2 on one side of an = sails over and changes to a $\surd$ sign on the other side

5.1.3 Cancelling

This applies to the removal of identities like $a-a$ or a/a. Some ways used to depict this were:

- A lightning bolt zaps the two terms and they disappear.
- An eraser appears and erases the terms.
- The two terms turn red and fall off the bottom of the screen together.
- A video game style spaceship flies in and fires missle to explode the term.
- A Monty Python style foot stomps out the terms.
- The Hand of God touches the term and it becomes a puff of smoke. This was used in the program that derived Kepler's first law (orbits are ellipses) from Newton's laws. The program made comparisons between the accomplishments of mathematics and physics and the accomplishments of art, drama, and music. Art was represented by the Sistine Chapel of Michaelangelo with the hand of God givin life to Adam. The essential cancellation in the math that makes the derivation work is r^2/r^2, this is done by the hand of God too.

- Multiplication sign snipping out term. The expression $\mathbf{v} \times \mathbf{v}$ is identically equal to 1. When this appears, the cross product sign magnifies around the surrounding $\mathbf{v}$'s and then squashes rapidly in y, snipping out the terms.

- Simply fading the terms out. This is, of course, the simplest and was done the most often.

5.1.4 Recalling old results

When a result from a previous program, or from a previous course is introduced some effort was made to indicate to the viewer where it came from. Some examples are:

- A Triginometry book flies in, opens, and trig identities fly out.

- A head with a hinged lid opens to recieve some intermediate results, later it returns and the intermediate results fly out.

- A hand pulls down a window shade with old energy equations.

- Reprise an entire scene from a previous program.

- Some results were derived against a background image of some distinctive color. Later, when the results are needed, a slide comes in containing the equation with the same background as old scene.

5.1.5 Substitution

Substitution involves taking an equation defining some variable and replacing occurrences of that variable into another equation. Some examples:

- Vertical Shrinking. A term is replaced with a number by shrinking the term vertically to zero and having the number expand up from zero in its place.

- Vacuum cleaner. The identity equation appears above the main equation. The replaced term from the lower equation moves up to the identity to merge with its copy there. The other side of the identity equation moves down to the empty spot left in the original equation.

- Several calculus identities (such as turning dr/dt into v) were shown by rotating the dr/dt about the y axis and having it become v when you see the other side.

- A term shrinks in x into a vertical line, then restretches out as new value.
- Lightening flash from identity equation to other equation.
- Whole identity equation crumples into substituted term. This crude, brute force approach was used in later programs to show the general structure of a derivation without clogging up with details.

5.1.6 Varying parameters

Some representation of inversely varying parameters was done by showing the equation

$$AB = constant$$

and having the letters for A and B scale up and down in opposite phase with each other.

5.1.7 Simplicity

The derivation of Kepler's law of elliptic orbits from Newton's $F = ma$ and the inverse square gravitational field is a fairly lengthy process that is not usually found in textbooks. We wanted to show this in all its detail so we spent some time fiddling with the derivation to make it as simple as possible.

Also the relation between orbital position and kinetic/potential energy at first looked forbidding, but we found a way to simplify it.

In general, some complex algebraic derivations motivated us to find better derivations than had been around before.

5.1.8 Jokes

The program on wave motion shows some approximate relations between wave speed and various physical parameters. The $\approx$ sign ripples like a propagating sine wave while these equations appear. This was done by modelling the lines of the $\approx$ sign with a one-cycle helix. Rotating it about x and then scaling by 0 in z made it ripple (see section on modeling with transformations in part IV).

5.1.9 Calculus

A few algebraic operations on calculus notation

- $\frac{d}{dt}$ flies in from left and impacts f to form $\frac{df}{dt}$.

- The f slides up and down to form $\frac{d}{dt}f$ from $\frac{df}{dt}$
- The two symbols $\int$ and dt move in on either side of f and clamp it together to form $\int f(x)dt$.
- A simple differential equation like $\frac{dx}{dt} = y$ is solved by moving dt to the other side to make $dx = y\,dt$. Then the left hand d hops over the equal sign and changes into a $\int$ sign, to make $x = \int y\,dt$.
- Integration is done by the $\int$ sign ratchetting across an expression, sort of like a credit card imprinter.
- $\oint$ is formed by drawing the circle on the $\int$ as the path of integration is traced out in geometric diagram in the background.
- $\oint\oint$ is formed by revealing the circle on $\int\int$ as a Gaussian surface is spread out around a volume in a parallel diagram.

5.2 Calculus

5.2.1 Limits

Use explosion to express the limiting process when Δ turns into d. The explosion was generated by a simple 2D pattern scaled up and faded out simultaneously.

5.2.2 Symbolic derivative machine

Since we evaluate derivatives and integrals symbolically many times in the series, we developed a quick way to do it—the derivative machine.

Design

The derivative machine is an expression transformer, it has two functions differentiation and integration. An expression goes in one end and comes out the other end. so it needed to be thin in the x direction so there would be plenty of room on each side to show the inputs/outputs. When the DM is first introduced, it comes in a crate marked "ACME Derivative Machine" (a hat tip to the old Chuck Jones Roadrunner movies). A crowbar shaped like an integral sign opens the crate.

Some random wheels and lights made it look Rube Goldberg-ish.The sides are not exactly straight and the wheels are not exactly round.

Internals

When the DM is introduced, in program 3, the internals are shown two ways.

As various elementary operations are introduced they shrink down into a sort of circuit board that is plugged into the machine, the door slams, and a new light blinks on on the front panel.

An alternative view of the internals was given briefly, showing the details of how the elementary operations are applied to take the derivative of the simple expression x^2. This was intended to be somewhat a metaphor on how symbolic derivative computer programs work. The input function comes in on a conveyor belt. An eyeball on a stalk comes down and looks at it. (This is indicated by a dotted line running fromthe eyeball to the function). This is the pattern recognizer. The derivative operation is basically one of matching the desired function against a list of known patterns which are pulled down into the scene like window shades. When the proper pattern is found and checked, there will be some dummy parameters in the pattern which need to be filled in with the specific terms from the equation. The eyeball observes these and some handles come down and simultaneously turn all occurrences of the dummy parameter into the specific term needed. Identities such as $x + 0$ or $x * 1$ are removed by an eraser. The expression $x + x$ is turned into $2x$ by a vise-like adder. The final expression is carried out on the conveyor belt.

Operation

The lever on the top controls the operation of the Derivative Machine. When you throw the lever to the right, it takes an expression in the left hopper and spits the derivative out the right hopper. When you throw the lever to the left it takes an expression in the right hopper and spits out the anti-derivative (integral) on the left. Sometimes the expression stays put and the DM passes over it. Note, it doesn't evaluate integral expressions, it just takes the anti-derivative (i.e. you dont feed $\int x^2$ in to get $\frac{1}{3}x^3$, you just feed in x^2.) As it operates, the horizontal and vertical scales cycle up and down a bit to give it a squash and stretch look.

5.2.3 Geometric derivative machine

Another representation of derivatives is graphical. When a function and its derivative are ploted above/below each other, the derivative curve is traced out by a mechanism that is a sort of geometric derivative machine. A tangent line slides across f with a spot at the point of tangency. A vertical line extend upward from the spot. A unit length horizontal line

runs to the tangent line. The resulting triangle measures the slope of the tangent. The unit distance in x direction causes the length of the y line to equal the derivative. The y piece is the 'derivative color', which is echoed on the derivative plot, and traces out the derivative curve.

5.2.4 Colors of Areas

Integrals are sometimes shown as areas under the curve of a function. Here the function is plotted against a dark yellow or red background. The area filled in under the curve is colored the appropriate "derivative color", i.e., green under a yellow curve, yellow under a red curve.

5.3 Inertia

The easist way to show inertia is just to animate things moving in uniform velocity. I used the Asteroid game as such a metaphor (although many MU people had never heard of the asteroid game.) This is a beneficial result of video games, we may have a generation of students with a real intuitive feel for inertia. Unfortunately, the game is now pretty obsolete. Newer games do not seem to have this effect.

5.4 Newton's Laws

Whenever a relation between position, velocity and acceleration is shown, the acceleration arrow is shown tugging on the velocity arrow, which is shown tugging on the position arrow. This gives a feel for integrating Newton's laws.

5.5 Vectors vs. Scalars

Vectors are arrows, of course. Ideally, scalars should be Represented by some shape that doesnt have any directionality. Different sized, filled circles would be ideal. The problem is that magnitudes cannot easily be compared, and there is confusion about area vs. radius as the representation. Could use speedometers, but there's an arrow for the indicator that might make it look like a vector. One can go crazy trying to out think viewers on what they will be confused by. We used bar graphs.

5.6 Vector Fields

We had extensive dealings with vector fields, mainly representing electric and magnetic fields and hydrodynamic velocity flow fields. The basic concept to be illustrated is that each point in space has associated with it a vector $\mathbf{E} = \mathbf{E}(x, y, z)$.

5.6.1 Representation

When vector fields are introduced they are shown with a hand moving a test charge around and the resultant force vector growing, shrinking and changing direction appropriately. The problem is how to show the essence of the entire field in one picture. There are two traditional ways to do this that were used in MU. Each of them raised problems and yielded some interesting insights.

Trail of Arrows

You can represent a vector field by placing a vector arrow with its tail at x, y, z and its magnitude and direction having the value of the field there. There are two problems with this:

1. The tail of the arrow is fixed at the location tested, but most of the bulk of the arrow is at some distance from the point it refers to. This sometimes gives a lop-sided appearance to a fairly symmetric field.

2. Scale. For an inverse square field, some of the arrows are enormous and others too short to see. Most textbooks exaggerate the lengths of the arrows to fit them into the diagram. We could do this but it might be confusing, again since it is a *computer* doing it, one expects numerical accuracy.

Field Lines

Another classic technique is to display field lines. These are defined as lines, starting at "sources" (e.g., positive charges), curving toward "sinks" (e.g., negative charges), and staying tangent to the direction of the field at each point. The strength of the field is represented by the spatial density or closeness of the field lines.

A note about rendering. Depth cueing is important to make complex collections of lines easy to interpret. The opacity of the lines is what is depth cued. This runs from maximum, at the *near* clipping plane, to minimum, at the *far* clipping plane. To make this show up, you must make these

planes as tight a bound on the scene as possible. If *near* is too near, or *far* is too far, the depth cueing will be to subtle. If the scene move in, or pulls back from the object being viewed it is necessary to animate the location of the near and far planes to keep a tight boundary. Also the lines are drawn sorted in Z so nearer (opaquer) lines overlay farther more transparent lines. Finally the brightness of the lines is gradated across the line to give a more solid "cylindrical" effect. (Using real cylinders here would look bad, they get bigger and smaller with distance and so look too "solid".)

Introducing lines When this concept is introduced, a "fur" of lines actually grows out of the positive charges, continuing to extend themselves until they hit negative charges, or go off to infinity. At each frame, the lines all terminate at a surface of constant electric potential. This threshold potential is animated as a linear function of frame number.

Once a field line is drawn, there is no longer an obvious visual key as to the *direction* of the field along a line. On occasions where this is important arrowheads are included. Some other ideas were toyed with, like cycling some dots along the lines, but these were rejected as making the image too complex.

Placement of Field Lines

A particular line can be generated numerically (see next chapter) from any given seed point. The real problem is, where to place the seed points. The placement of seed points must be arranged to make the spatial density of lines properly represent the strength of the field.

2D vs./ 3D Here we encounter some difficulty with many textbook diagrams that simplify the problem to 2D. For example, consider a point charge. An obvious distribution of the field lines in 2D is to make them equally spaced radially about the point. If you now place two opposite charges on either side, but unequally spaced along a straight line there are problems. The wrong number of lines connects from the center to each of the side charges. This was very perplexing until we finally referred back to Maxwell's original book and realized that the entire situation has cylindrical symmetry in 3D. The lines must *start out* equally spaced in 3D in order work properly. This means they must be *unequally* spaced radially in the 2D plane, anticipating an effective rotation of the 2D diagram about the axis of symmetry in order to make the 3D diagram.

3D placement The easiest way for the field line density to be correct in 3D is to begin the seed points at regions of known, simple geometry and

field strength. Then their properties, and the properties of space, will keep them at the proper density. Some examples of simple starting situations are:

Constant Field Take a plane perpendicular to the field. Pierce it with a regular grid of field lines. The spacing must be proportional to the square root of the field strength.

Point Charge Start near point charges and use a radially symmetric set of seed points. Generating a set of radially symmetric points around a point is an interesting geometric/topological problem. One can use, for example, the vertices of a regular polyhedron (as was done here). The problem comes in a scene where there are two charges, with one twice the size of the other. Given a set of points about the smaller charge, how do you make exactly twice as many points symmetrically about the larger charge (regular polyhedra only come in 4,8,6,12,and 20 vertix versions).

Magnetic Field around Wire Consider the magnetic field around a current carrying wire. The field strength is inversely proportional to radial distance from the wire. The field direction is perpendicular to the wire. A particular field line is just a circle surrounding the wire and perpendicular to it. The question is, what should the radii be?

Consider a plane containing the wire. The lines are all perpendicular to this plane. Consider the spacings of the field lines through a rectangle in this plane. The *magnetic flux* through the rectangle is the integral of the field strength over that rectangle. The number of lines is proportional to the magnetic flux. Integrating the $1/r$ field between two radii gives a flux that goes as the logarithm of the radius. Given a rectangle stretching radially from r_0 to r_1, the number of field lines is

$$N = k(\ln r_1 - \ln r_0)$$

or

$$r_0 e^{N/k} = r_1$$

Plugging integer steps for N shows that the radial spacing of the circular field lines should go exponentially. That is, the radii of two successive circles should have the same ratio.

This technique can be applied to other circularly symmetric fields such as dipole fields. The general technique is

1. Integrate field strength as a function of radius, times differential area.

2. Invert this function, giving r as function of N.

3. Mark off equally spaced units on N axis to give desired radii on r axis.

This was done numerically for many situations, such as the magnetic dipole, which are analytically complex.

Note that $\mathbf{B} \rightarrow \infty$ as you get closer to the wire, the number of lines gets infinite there. Contrast this with $\mathbf{E} \rightarrow \infty$ as you get closer to a point charge, the number of lines is constant, just density goes to ∞.

5.6.2 Dynamic Fields

What happens when the field strength is changing with time? How should the seed points move as functions of time?

In fact, you don't even have to consider time. The problem arises even with static fields. Since anywhere you *start* a set of field lines is as good as anywhere else, a perfectly consistant picture is made by any field line collection that just satisfies the density constraints on each individual frame. Each frame could have a different distribution, each of which individually satisfy the density constraint. This is obviously a problem, we shouldn't introduce motion where there is none.

Consider some particular cases.

Constant direction, Varying magnitude

The lines are on a regular grid, scaled by $1/\sqrt{F}$. This means that the center of scaling doesn't move, the field lines seem to 'breath' in and out.

AC Solenoid field

Consider the magnetic field of a solenoid with applied AC. For a given current (B field strength) there is no indication of field strength except as distribution of seed points. The shapes of lines will be same no matter what strength of field. Starting points generated by integrating field strength across face of magnet.

Electromagnetic Plane Wave

Consider the plane containing the propagation direction and perpendicular to the electric field. The field strength is (ignoring some constants)

$$E = k \sin(x + t)$$

At a given time, the field lines should be in dense rows where sin is large, and be absent from rows where sin is zero. As time progresses, a dense row could 'breathe out' getting sparser and sparser until it's empty. Simultaneously a nearby empty row would 'breathe in'.

While this is a perfectly consistant representation, it seemed to complex. We just had the rows themselves translate forward along the direction of propagation.

Capacitor field

How about field lines between plates in a capacitor. Made of a bent spiral so that as scale changed (breathing mode) proportionally to $\sqrt{\mathbf{E}}$ field lines drop off the edges of the plates at a uniform rate.

Induction

B field lines going thru a conducting loop as **B** field collapses. Topology changes. Life gets complex.

5.6.3 Special Problems with Electromagnetic Fields

There is an additional problem with animated illustrations of electrostatic fields. You are tempted to show the shape of the field by "flying around" the field lines. With electrostatics, the field lines don't change. If *either* the charges or the observer move, there are electrodynamic effects; ripples move along the field lines.

This can be explained away or minimized by carefully picking time and distance units, but it's the overall effect that worried me.

But a static view of 3D field lines is not as comprehensible as being able to fly around it. I finally did some gradual rotation of the view, or rocking back and forth, but avoided too violent motions.

5.6.4 Remaining problems and ideas for further work

I feel we have just scratched the surface of this subject. Just thinking about the problem of correct placement of seed points and about where they imply the field lines will go is a very interesting process. Some situations were still too complicated to do correctly.

As an example, consider the field of a planet like Uranus interacting with the Interplanetary field generated by the sun. Correct spacing of field lines about surface of planet. As the planet rotates the topology changes.

5.7 Vector Calculus

5.7.1 Derivatives

Derivatives of vectors. geometric construction

5.7.2 Line Integrals

Inital nudge with finger in + and − directions. Then continuously accumulate integral as finger pushes point throught smooth path. Finger always points tangent to path and represents $d\mathbf{r}$.

Integrating ΔW (work) for test charge, the finger moved an actual charge. Integrating ΔV (voltage) there was no test charge. Finger just moved around path.

5.7.3 Surface Integrals

Color in area with fist as $d\Phi = \mathbf{F} \cdot d\mathbf{A}$ accumulates. Show number of lines, adjusting color and opacity of surface so intersections are visible. Puffs of smoke as they pierce. Integral over closed surface shows punctures, (could do better since no distinction between + and - for entering/exiting.)

5.8 Wave interference

The program on optics deals mostly with wave interference. To show constructive and destructive interference it is necessary to be able to visually add two waves. The problem comes in that the values can be negative. Solution was to make medium gray stand for 0, black for -1 and white for +1. A null situation is a medium grey field. Waves are alternating black and white ripples moving through. They add nicely and interference looks natural. We actually used yellow instead of white to give the impression of electric field intensity (yellow).

Also this was used in illustrating DeBroglie atom model. Instead of showing the electron wave as a geometric distortion permpendicular to the plane of a circular orbit, it was shown as alternating black and white regions on a grey background. Grey was selected since we are representing sort of a probability density.

5.9 Thermodynamics

5.9.1 Ideal gasses

Show constant gravitational force on piston as constant pink (force) arrow. Show impacts on piston as force (pink) arrow that fluctuates. This will be the sum of all Δmomenta of atoms colliding with piston during last frame time. Show momentum of piston as pale yellow arrow, integral of sum of forces. Yellow slowly grows downward until an impact. Piston and arrows dance around with brownian motion.

Used dimmer, earth tone colors used for the cylender and piston to make atoms stand out. Also to give the impression of mass and to make them look like impenetrable barriers.

Streaks on atoms to show velocity. Streaks are 4 timesetps long.

5.9.2 Non-ideal gas, Lennard-Jones potential

2D Lennard-Jones simulation with yellow momentum vectors. Had to adjust color of atoms so vectors were visible.

5.9.3 PVT diagrams

Thermometer labeled with colored regions for solid, liquid and gas. Colored region expands out to plane showing how boundaries a function of Pressure also. Full PT diagram with brown/white/blue regions for states of matter. Blue color represents phase difference between liquid and gas. It is actually a plot of volume V as solved for in van der Waals' equation.

5.9.4 Piston engines

Work meter on carnot cycle/ heat engine. Shows 4 cycles of carnot cycle necessary to get 1 nodule of energy out.

5.9.5 Heat flow diagrams

Heat flow shown as cycling blue (energy) arrows. Width of arrow is size of quantity flowing. Widths of energy input arrow equals sum of widths of energy/heat outflow arrows. Entropy flow shown as cycling red (entropy) arrows.

5.10 Relativity

5.10.1 Michaelson-Morley experiment

Mirror apparatus shown with photon bouncing along path. Grid background shows hypothetical 'ether' coordinate system. Photon moves at constant rate relative to grid, not mirrors. Timing varies on thru leg vs perpendicular leg, and when rotate apparatus.

Temporal optical illusions in motion of light with respect to grid.

Show diffraction pattern as two photons merge at screen. Light if arrive at same time, dark if arrive at different times (somewhat metaphorical).

5.10.2 Cartoons

Cartoons of Einstein and Lorenz used to show difference in two observers measurements. Started with Galilean transformation by showing Galileo on flatcar. Then showed Lorenz transformation by showing Lorenz on flatcar.

Albert and Henry characters drawn by Mike Shaw. Digitized various heads and various body parts separately. Animation software allows for selecting which of several heads and bodies to use for a given frame.

Colors for Albert (tan) and Henry (blue). These originally came from the colors of their suits. Colors also show up in coordinate grids, color of spaceships and pool balls.

Timing of light sphere adjusted to make stuff fit on screen

Positioned events near horizon since this is really a one (spatial) dimension problem.

Fundamental paradox: light moves at same speed, even for relatively moving observers. Better way to do it: thought baloons conaining "v=c" for Albert and Henry watching light beam.

Note direction of Lorenz contraction of poolballs (almost did this wrong)

5.10.3 Space-Time Diagrams

Space Time diagrams are fairly conventional representations of these things but they look nice as shaded polygons. They were made by scaling and skewing a simple extrusion of the outlines of the cartoon characters. They were made transparent so viewer could see whole diagram from one view.

Small Yellow sphere marks spacetime events.

Use of blue green background colors to show time slice thru spactime diagram.

Had to squash out z depth of spactime diagram when it turns to grid to avoid confusion due to parallax.

Note dimension lines measuring A distance and H distance and how perpendicular to arrow goes along the appropriate observers grid.

5.11 Quantum Mechanics

Spiral electron path (UV catastrophe). What is shape of spiral? What direction does light move in relation to electron position. Parallel or perpendicular?

Bohr atom emitting/absorbing light. How does this look? Commonly shown as little particle or ripple moving away from atom. But the wavelength of the light is much greater than the size of the atom. Also speed of light = 137 times classical tangential speed of electron in lowest orbit. Given 1 second period of electron, whole wave passes away in 1/4 frame time. One way to show would be to make whole screen flash for one frame. But this it too fast and you can't easily tell the difference between emission and absorbtion. Also what about directionality? Atom is symmetrical in correct QM description. Even though we are showing semi-classical approximation (with the electron position giving some directionality), it didn't seem right to show a particular direction that light goes in. Cheat by showing fuzzy ring of light moving out/in in about 10 frames.

Stairstep for energy levels of H.

Individual dots making diffraction pattern. Important to make each dot distinct.

Electron clouds for hydrogen orbitals. Each pixel is integral of electron probability along line of sight. This looks real fuzzy. Attempt to show structure by having cube slice across to show various cross sections. On the surface of the cube, the integral is brighter to emphasize cross section.

Chapter 6

Mathematical Models

This chapter is devoted to the mathematics behind the various physical simulations. It will deal just with the physics and numerical techniques. Implementation details such as data formats, command language interfaces, etc. are left for a later chapter.

A lot of the physics that we showed was illustrated by examples that didn't require very sophisticated simulation. In most cases it just required plugging a value for *time* into some closed form algebraic equation. Others required numerical integration of force laws. Here follows an assortment of various techniques.

6.1 Falling body

Equation of motion is just

$$y(t) = y_0 + v_0 t - \frac{1}{2} g t^2$$

Most of the time $v_0 = 0$ so its just a simple parabola.

6.2 Momentum transfer in pool balls

Consider two balls of radius 1 and equal masses. The momenta before the collision are $\mathbf{p}_{1b}$ and $\mathbf{p}_{2b}$. The collision imparts equal and opposite changes in momentum to each ball, directed in the line connecting their centers at the time of impact. Calling this direction $\hat{\mathbf{r}}$, we have after the collision:

$$\begin{aligned} \mathbf{p}_{1a} &= \mathbf{p}_{1b} + \alpha\hat{\mathbf{r}} \\ \mathbf{p}_{2a} &= \mathbf{p}_{2b} - \alpha\hat{\mathbf{r}} \end{aligned}$$

Conservation of kinetic energy can be written in terms of momentum since, for any object:

$$\mathbf{p} \cdot \mathbf{p} = m^2 v^2 = 2mK$$

so

$$\mathbf{p}_{1b} \cdot \mathbf{p}_{1b} + \mathbf{p}_{2b} \cdot \mathbf{p}_{2b} = \mathbf{p}_{1a} \cdot \mathbf{p}_{1a} + \mathbf{p}_{2a} \cdot \mathbf{p}_{2a}$$

Plugging in the above equations for momenta after the collision and solving for α

$$\alpha = -\hat{\mathbf{r}} \cdot (\mathbf{p}_{1b} - \mathbf{p}_{2b})\hat{\mathbf{r}}$$

In the particular case animated, the balls are arranged to collide along the 45 degree diagonal, and ball 2 starts out at rest. In other words (giving names to components of $\mathbf{p}_{1b}$

$$\begin{aligned} \mathbf{p}_{1b} &= (a, b) \\ \mathbf{p}_{2b} &= 0 \\ \hat{\mathbf{r}} &= \frac{\sqrt{2}}{2}(1, 1) \end{aligned}$$

Plugging in leads to

$$\begin{aligned} \mathbf{p}_{1a} &= \left(\frac{a-b}{2}, \frac{-a+b}{2}\right) \\ \mathbf{p}_{2a} &= \left(\frac{a+b}{2}, \frac{a+b}{2}\right) \end{aligned}$$

Notice that the two final vectors are perpendicular.

An even easier derivation would have been for the collision to be along the x axis, with ball 2 starting at rest.

$$\begin{aligned} \mathbf{p}_{1b} &= (a, b) \\ \mathbf{p}_{2b} &= 0 \\ \hat{\mathbf{r}} &= (1, 0) \end{aligned}$$

Plugging in then leads to

$$\begin{aligned} \mathbf{p}_{1a} &= (a, 0) \\ \mathbf{p}_{2a} &= (0, b) \end{aligned}$$

Notice that the two final vectors are perpendicular.

The whole scene can be rotated to get other collision geometries.

6.3 Vector Field Sources

To generate field arrows or field lines we must be able to compute the field vector at any desired point in space, x, y, z. The fields we dealt with come from two sources, (1) collections of point charges generating a static electric field, (2) collections of finite-sized current loops generating a static magnetic field.

6.3.1 Static Electric Fields

An electrostatic field vector is just the sum of the contributions of each point charge. That is

$$\mathbf{E} = \sum_i \frac{q_i}{r_i^2} \hat{\mathbf{r}}_i$$

where

$$\begin{aligned} q_i &= \text{charge on point charge } i \\ \hat{\mathbf{r}}_i &= \text{unit vector to point charge } i \\ r_i &= \text{distance to point charge } i \end{aligned}$$

A constant multiplicative factor of $1/4\pi\epsilon_0$ is neglected here since the field line integration process is insensitive to a global change in the magnitude of the field.

The field routine simply maintained a list of up to 20 point-charge locations and looped through this list when given an x, y, z.

6.3.2 Magnetic Dipole Field

The magnetic field generated by a current loop is not so simple. There is no analytic formula for calculating the field vector at any desired point in space. The field can be found by numerically integrating the Biot-Savart law, or by evaluating elliptic integrals. I tried each approach because I didn't know which would work better. (In the end they seemed to work about the same).

We first find the field in a simple geometry that eases the algebra. The current loop is assumed to lie in the xz plane, centered at the origin, and have radius a. We wish to evaluate the field at a point above the x axis at location $(\rho, y, 0)$.

Biot-Savart Law

This law states that an incremental contribution to the magnetic field $d\mathbf{B}$ due to a small snip of wire $d\mathbf{l}$ is given by

$$d\mathbf{B} = I\frac{d\mathbf{l} \times \hat{\mathbf{r}}}{r^2}$$

where

$$\begin{aligned} I &= \text{current through wire} \\ \hat{\mathbf{r}}_i &= \text{unit vector to wire} \\ r_i &= \text{distance to wire} \end{aligned}$$

Again we are neglecting global constant factors.

It remains to integrate small segments $d\mathbf{l}$ around the current loop to find a net value for $\mathbf{B}$. We can apply symmetry arguments to show that the z component of $\mathbf{B}$ is zero so it need not be calculated. Also the contributions of each wire snip on the positive and negative z halves of the loop are equal, so these calculations can be merged. Sample code to do this after factoring as much algebra out of the loop as possible is

Inputs: RHO,Y,NSEGS,A

```
B = 3.14159/NSEGS
XLOOP = A*COS(B/2.)
ZLOOP =-A*SIN(B/2.)
RLOOP2 = A*A
FACTOR= RHO*RHO + A*A + Y*Y
BX=0;   BY=0
FOR I=1 TO NSEGS
   RSEG2 = -2*XLOOP*RHO + FACTOR
   RCUBED = RSEG2+SQRT(RSEG2)
   IF (RCUBED ne 0)
      BX = BX + (-XLOOP*Y)   /RCUBED
      BY = BY   ( XLOOP*RHO - RLOOP2)/RCUBED
   TEMP  = XLOOP*COS(A) + ZLOOP*SIN(A)
   ZLOOP =-XLOOP*SIN(A) + ZLOOP*COS(A)
   XLOOP = TEMP
```

Elliptic Integrals

This integration has been done semi-analytically in a few advanced textbooks and written in terms of elliptic integrals. The two components of $\mathbf{B}$ in terms of a, ρ, and y are found as follows

Define

$$\begin{aligned} \Omega &= \sqrt{(a+\rho)^2 + y^2} \\ m &= \frac{4a\rho}{(a+\rho)^2 + y^2} \end{aligned}$$

Then

$$B_x = \frac{y}{\rho\Omega}\left(-K(m) + \frac{a^2+\rho^2+y^2}{(a-\rho)^2+y^2}E(m)\right)$$
$$B_y = \frac{1}{\Omega}\left(K(m) + \frac{a^2-\rho^2-y^2}{(a-\rho)^2+y^2}E(m)\right)$$

Where K and E are *complete elliptic integrals of the first and second kind* defined as

$$K(m) = \int_0^1 [(1-t^2)(1-mt^2)]^{-1/2}\,dt$$
$$E(m) = \int_0^1 (1-t^2)^{-1/2}(1-mt^2)^{1/2}\,dt$$

These are not analytically solvable, but can be approximated.

E is approximated by

```
AM1=1.-m
IF(AM1 >  0)
   A1=.46301 51;     A2=.10778 12
   B1=.24527 27;     B2=.04123 96
   E = (A2*AM1+A1)*AM1+1.
     + (B2*AM1+B1)*AM1*ALOG(1./AM1)
ELSE
   E = 1.
```

and K is approximated by

```
AM1=1.-m
IF(AM1 > 0)
   A0=1.38629 44;    A1= .11197 23;    A2= .07252 96
   B0= .50000 00;    B1= .12134 78;    B2= .02887 29
   K = (A2*AM1+A1)*AM1+A0
     +((B2*AM1+B1)*AM1+B0 )*ALOG(1./AM1)
ELSE
   K=99999999.
```

Stacking Them

The field evaluation routine maintains a list of current-loop locations, stacked up with varying y values, still parallel to the xz plane. This allows approximation to a solenoid field by stacking up about 10 loops of equal radius

and equal current. Then the net field at any desired x, y, z is

$$\mathbf{B} = \sum_i I_i (B_{x_i} \frac{x_i}{\rho_i}, B_{y_i}, B_{x_i} \frac{z_i}{\rho_i})$$

where

$$\rho_i = \sqrt{x_i^2 + z_i^2}$$

and the B_{x_i}, B_{y_i} are evaluated at ρ_i, and $y - y_i$.

6.4 Field Lines

The simplest vector field lines did not require complex modelling. Lines representing constant fields were just an array of parallel straight lines. The magnetic field near a straight current carrying wire is just a collection of concentric circles perpendicular to the wire. These types of field lines were just modelled explicitly. The spacings of the lines were modelled as described in a previous chapter.

Some simple hydrodynamic velocity fields were also discussed. These were not drawn as field lines, but rather by showing a collection of marker particles flowing with the field.

For more exotic shapes, the field lines needed to be numerically generated. A field line is defined as being everywhere tangent to the field vector. A differential distance $d\mathbf{s}$ along the field line can then be defined as some small fraction ϵ times the normallized field vector

$$d\mathbf{s} = \epsilon \frac{\mathbf{F}}{|\mathbf{F}|}$$

Note that the sign of ϵ tells which direction we are to move along the line.

6.4.1 Stepping Along

The simplest technique for tracing out a line is Euler integration. Start at some x, y, z. Evaluate $d\mathbf{s}$ at that point and add it to x, y, z. Repeat. The size of ϵ tells the geometric distance of one step.

This is not very accurate numerically. A much better integration technique is 4th order Runge-Kutta. It calculates a step as a weighted average of four Euler steps with various nearby derivatives. The Runge-Kutta formula for scalar functions appears in many numerical analysis books. The generalization to vector functions is best illustrated in terms of the function

```
FIELD(X,Y,Z,  FACTOR, FX,FY,FZ)
```

Which evaluates the field vector at `X,Y,Z` and returns it in `FX,FY,FZ` scaled by `FACTOR`, (this latter is to make the code cleaner). One step then looks like:

```
FIELD(X,Y,Z,   1,   FX,FY,FZ)
DELTA=EPSILON/SQRT(FX*FX+FY*FY+FZ*FZ)
FIELD(X       , Y       , Z       , DELTA, DX1,DY1,DZ1)
FIELD(X+.5*DX1, Y+.5*DY1, Z+.5*DZ1, DELTA, DX2,DY2,DZ2)
FIELD(X+.5*DX2, Y+.5*DY2, Z+.5*DZ2, DELTA, DX3,DY3,DZ3)
FIELD(X+   DX3, Y    DY3, Z    DZ3, DELTA, DX4,DY4,DZ4)
X = X + (DX1 + 2*DX2 + 2*DX3 + DX4)/6.
Y = Y + (DY1 + 2*DY2 + 2*DY3 + DY4)/6.
Z = Z + (DZ1 + 2*DZ2 + 2*DZ3 + DZ4)/6.
```

The second evaluation of `FIELD` can be eliminated by using the results of the first evaluation, but the above technique is shown for clarity.

Step Size

The step size is important. Too big and details will be lost, too small and you waste a lot of time and maybe build up round off errors. The step size can be dynamically adjusted by means of an error estimate. Effectively, the result of *two* steps at half the step size is compared to the result of *one* step at the current size. If these differ by more than some error tolerance the step size is reduced. If they are closer than some other tolerance, the step size is doubled.

Actually evaluating two half steps each time is wastefull. I got a semi-justified approximation to the error estimate by applying the Runge-Kutta formulas to the differential equation for a circle. This can be analytically solved and cast in terms of the partial steps we are already calculating. This results in an error vector

$$e = \Delta_1 - 2\Delta_3 + \Delta_4$$

where Δ_1 = `(DX1,DY1,DZ1)`, etc. Effectivly we are saying that, if the `(DXn,DYn,DZn)` are about equal, the error is small.

The square of the length of this error vector is used to trigger step halving or doubling with code like

```
IF(E2 > threshold    ) EPSILON=EPSILON/2
IF(E2 < threshold/512) EPSILON=EPSILON*2
```

This gives some hysteresis to the step size change.

When to stop

There are four criteria that were used to terminate the line generation. Two use the value of the "Potential", defined as the running integral of the magnitude of the field vector times the step size. This is the same as the electrostatic potential for electric fields. For magnetic there is no unique scalar potential, but the same integral works ok as a termination criterion. The four termination criteria are:

1. You move outside a maximum distance from the origin. This is to catch lines that go to infinity.
2. The accumulated electric or magnetic potential reaches some threshold. This threshold is animated to show the field lines growing out of a source and flowing into a sink.
3. The accumulated potential begins to decrease. This is to catch the line when it hits a point charge.
4. You get back within ϵ of the initial starting location. This is to terminate looping magnetic field lines.

6.5 Sum of potentials

6.6 Swirl of particles down drain

6.7 EM wave ripples in field lines (140)

6.8 Spiraling electron (149)

6.9 Fourrier synthesis (151)

6.10 Water (22)

6.11 Wave (22)

6.12 Galaxy contraction (20)

6.13 Surface Charge on Conductor

to calculate color on surface of contuctor

6.14 Ripple Tank Simulation

to make black/white ripples for light wave interference patterns

6.15 Hydrogen Wave Orbitals

The best quantum mechanical description of the Hydrogen describes the electron location as a probability cloud, a scalar function of x, y, z. This is often represented in textbooks by plotting the magnitude as functions of radius or angle, or by plotting the probability on some 2D slice as brightness.

I wnated to get a more 3D effect and generated an image of the three dimensional cloud. The brightness at each pixel is the integral of the probability along the line of sight, through the probability cloud. This tended to make pictures that were, well, blurry. In order to make the viewers orientation more clear, the clouds were surrounded by a box. Only the probability inside the box was included. This speeded up the per-pixel integration time by restricting the range in space over which integration was necessary.

Further improvements were obtained by pre-evaluating those parts of the probability function that depended only on radial distance from the nucleus and placing them in a table. The table is indexed by r^2 removing the need even to calculate square roots when looking up a value.

The input to the program consists of:

- A minimum and maximum value for each of x, y, and z for which integration is required. This defines a "bounding brick" of arbitrary position and x, y, z extent. (It is the responsibility of other standard rendering programs to draw the actual lines of the surrounding box.)
- A flag indicating the required orbital required

The program then executes as follows

1. Find the maximum possible value of r^2 from the max/min values of the bounding brick.
2. Fill the radial function table
3. Find the possible range of pixels affected on the screen from the max/min values and the current viewing transform.
4. For each pixel:

(a) Intersect the line-of-sight ray with the boundaries of the bounding brick. Find the max/min distance along the ray that is inside the brick. A special case was made if max=min. This allows the bounding brick to be zero thickness, enabling simple cross sections to be rendered.

(b) Divide this segment into samples for trapeziod-rule integration.

(c) Evaluate the x, y, z location in "Hydrogen Atom" space at those samples. Find r^2.

(d) Evaluate the probability function by combination of table look-up of radial part and some simple arithmetic on x, y, z components. Add them together.

The precise probability functions are given in the following table. Notice how the per-sample evaluation has been minimized. The middle column is the definition of entries in the table generated in step 2. Table entry i corresponds to radius R^2. The table is 512 entries long.

Orbital	Tabulated Function, T	Per-sample Evaluation
	$R = \sqrt{(i/512) * R^2_{max}}$ $E = e^{-R/n}$	$\rho^2 = x^2 + y^2$ $T = \text{table}(\rho^2 + z^2)$
$1s$	E^2	T
$2s$	$(2-R)^2E^2$	T
$2p_z$	E^2	Tz^2
$2p_{xy}$	$.5E^2$	$T\rho^2$
$3s$	$1.55^2(27-18R+2R^2)^2E^2$	T
$3p_z$	$2.83^2(6-R)^2E^2$	Tz^2
$3p_{xy}$	$2^2(6-R)^2E^2$	$T\rho^2$
$3d_{m=0}$	$.816^2E^2$	$T(2z^2-\rho^2)^2$
$3d_{m=1}$	2^2E^2	$Tz^2\rho^2$
$3d_{m=2}$	E^2	$T\rho^4$

The funny scale factors in the tabulated functions are to provide some normalization. A global scale factor was applied to the probability integrals in order to scale them to a proper 0...1 intensity range. The value of this scale factor was adjusted empirically.

6.16 PVT Diagram

The ideal gas law is

$$PV = nkT$$

A better approximation to a real gas is provided by the van der Waals equation

$$\left(p + \frac{a}{V^2}\right)(V - b) = nkT$$

When showing a phase diagram (plotting pressure P against temperature T) the volume V can be used to show regions of liquid (low V) versus gas (high V). The Van der Waals equation gives a cubic equation to solve for V at any desired P and T. In most places this has only one solution, but in some regions there are three solutions, corresponding to the phase transition of liquid to gas. A large V solution represents gas, a small V solution represents liquid, and an intermediate V represents a state of meta-stable equilibrium. This is actually a catastrophe surface, but was plotted to show an abrupt phase change by selecting one of the two extreme solutions. Plotting P against V for constant temperature yields

$$P = \frac{nkTV^2 - aV + a}{V^3 - bV^2}$$

This function looks somewhat like a cubic, having a local max and a local min. The phase transition happens at a pressure P_0 and a voume V_0 that causes

$$\int_{V_{min}}^{V_{max}} P(V) - P_0\, dV = 0$$

where V_{min} and V_{max} are those volumes at which $P = P_0$. This is all rather elaborate, requiring solving a cubic polynomial explicitly.

6.17 Kepler orbits

Proving that particles moving under an inverse square gravitational force is fairly easy. Finding an explicit equation for the position as a function of time is harder. This solution is best given in polar coordinates. The polar equation for a general conic is

$$r = \frac{p}{1 + e\cos a_T}$$

where a_T is an angle called the *True Anomaly*. We wish to find this angle as a function of time.

I will omit derivations and just give the punch line. The true anomaly is calculated in terms of two intermediate results, the *Mean Anomaly*, a_M, and the *Eccentric Anomaly*, a_E. The mean anomaly is an angle that progresses linearly with time, going from 0 to 2π in one orbital period. The relation between these quantities depends on the eccentricity.

For an ellipse ($0 < e < 1$)

$$\begin{aligned} a_M &= a_E - e \sin a_E \\ \tan \frac{a_T}{2} &= \sqrt{\frac{1+e}{1-e}} \tan \frac{a_E}{2} \end{aligned}$$

For a hyperbola ($1 < e$)

$$\begin{aligned} a_M &= -a_E + e \sinh a_E \\ \tan \frac{a_T}{2} &= \sqrt{\frac{1+e}{e-1}} \tanh \frac{a_E}{2} \end{aligned}$$

The general technique is, then, from the time t and period T calculate mean anomaly $a_M = 2\pi \frac{t}{T}$. Convert this to eccentric anomaly; convert that to true anomaly.

There is a bit of a problem in that the equation linking a_E and a_M (which is known as Kepler's equation) has no analytic solution for a_E and must be solved numerically. Newton iteration works well (I'm sure he would be pleased).

The calculation of a_T from a_M and e was done so often that it was added to the animation program algebraic expression evaluator.

6.18 Ideal Gas

An ideal gas consists of a number of point-like atoms with various velocities enclosed in a box. The atoms do not interact with each other, only with the walls of the container. We illustrate the interactions between temperature, pressure and volume by placing the ideal gas in a cylinder with a moveable piston. The walls of the cylinder are rigid, but the piston is free to move. It has a constant downward accelleraton due to gravity, and fluctuating upward accelleration due to collisions with the individual atoms. Modelling the interaction can be broken down into several parts.

6.18.1 General Simulation Strategy

In between collisions, each atom moves with constant velocity. The piston falls under gravity with constant accelleration. This continues until the next collision of an atom with either the walls or piston.

Collisions with the side walls do not materially affect the simulation. The horizontal coordinate of each atom's position moves back and forth at whatever its initial horizontal speed, with the speed flipping sign on each

collision. The horizontal positions are therefore almost just decoration to the simulation.

Collision with the piston is a different matter. When an atom collides with the piston, there is some energy and momentum transfer. It will, in general, recoil with a different velocity. After the collision, the piston itself has an altered velocity, so its further collisions with other atoms are modified.

The next atom-piston collision will take place at some time in the future. The extrapolation of atom position to piston collision means that the atoms which are moving downward must be bounced off the floor on their way to the collision. This is most easily accomplished by actually extrapolating to the next "interesting event" defined as the earliest of

- The time of the next atom-piston collision
- The time of the next atom-floor collision

The simulation is then be carried out in the following manner:

1. The motion of each atom and the piston are extrapolated into the future to see when the next 'interesting event' occurs.
2. The position of the piston and of all atoms are moved forward in time by this amount.
3. The interaction of the interesting atom with either the floor or the piston is calculated, updating the velocities of each.
4. Goto step 1.

6.18.2 Next Interesting Event

Collision with piston

The motion of the piston until the next collision is

$$y_p(t) = y_p + v_p t + \frac{1}{2} a_p t^2$$

where $a_p = -g$ =accelleration of gravity. The motion of atom i until the next collision is

$$y_a(t) = y_a + v_a t$$

Solving these two equations for t involves solving the quadratic equation

$$t^2(\frac{1}{2}a_p) + t(v_p - v_a) + (y_p - y_a) = 0$$

Collision with floor

Given the floor at coordinate $y = 0$, the time until atom i impacts the floor is

$$t = \frac{-y_a}{v_a}$$

Putting it together

The atom list is scanned. For each atom, calculate the two solutions to the quadratic and the one solution for floor bounce. Maintain a pointer to the atom with the smallest future Δt so far. A new Δt is only a candidate for comparison with the current one if it is positive.

Avoiding Bouncing an Atom Twice

There is a bit of a numerical problem with this approach that requires some flags. The 'interesting atom' that was updated on the *last* main iteration will generate two solutions to the *current* piston-collision quadratic, one of which is $\Delta t = 0$ (the one we just did). Round-off error may cause this to be slightly positive, giving a false entry for the next Δt. This can be avoided by remembering the *name* of the last interesting atom, and a *flag* for what it hit (floor or piston). A new interesting Δt is discarded if it comes from the same atom, hitting the same thing, as last time.

Code fragments to find next atom and time to collision.

```
TNEXT=99999999.
NEXTA=0
KLAST=KEVENT
FOR I = 1 TO NBRATOMS
    IF(I=LASTA and KLAST=1) skip this one
        SQUAD(AP/2., VP-VY(I), YP-Y(I), T1,T2) "piston"
        IF(0<T1 AND T1<TNEXT)
            NEXTA=I;  TNEXT=T1;  KEVENT=1
        IF(0<T2 AND T2<TNEXT)
            NEXTA=I;  TNEXT=T2;  KEVENT=1
    IF(I=LASTA and KLAST=2) skip this one
        IF(VY(I)ge0) skip this one              "floor"
        T1=-Y(I)/VY(I)
        IF(T1<TNEXT)
            NEXTA=I;  TNEXT=T1;  KEVENT=2
```

6.18.3 Advancing Time

Code fragment to update all objects by DTIME

```
YP = YP + VP*DTIME + .5*AP*DTIME*DTIME  "piston pos"
VP = VP + AP*DTIME                      "piston vel"
FOR I=1 TO NBRATOMS
    X(I) = X(I) + VX(I)*DTIME        "Move atom"
    Y(I) = Y(I) + VY(I)*DTIME
    IF(X(I)<0)
        X(I)= -X(I);  VX(I)=-VX(I)   "Bounce left"
    IF(X(I)>1)
        X(I)=2-X(I);  VX(I)=-VX(I)   "Bounce right"
```

6.18.4 Collisions

With Walls

Collisions with the walls and floor are just perfect reflections. They just reverse the sign of the velocity component corresponding to the wall hit.

With Piston

An elastic collision must conserve both momentum and kinetic energy. Let the masses of the piston and atom be:

$$\begin{aligned} M &= \text{Mass of piston} \\ m &= \text{mass of atom} \end{aligned}$$

Let the initial velocities be

$$\begin{aligned} (0, A) &= \text{Initial velocity of piston} \\ (B, C) &= \text{initial velocity of atom} \end{aligned}$$

Let the final velocities be

$$\begin{aligned} (0, a) &= \text{Final velocity of piston} \\ (b, c) &= \text{final velocity of atom} \end{aligned}$$

We wish to solve for the unknowns a, b, c in terms of the known initial conditions M, m, A, B, C. Conservation laws give three equations to solve for the three unknowns. Conservation of horizontal momentum, vertical

momentum, and kinetic energy give, respectively

$$\begin{aligned} Bm &= bm \\ AM + Cm &= aM + cm \\ \frac{1}{2}MA^2 + \frac{1}{2}m(B^2 + C^2) &= \frac{1}{2}Ma^2 + \frac{1}{2}m(b^2 + c^2) \end{aligned}$$

Solving these equations yields

$$\begin{aligned} a &= \frac{M-m}{M+m}A + \frac{2m}{M+m}C \\ b &= B \\ c &= \frac{2M}{M+m}A - \frac{M-m}{M+m}C \end{aligned}$$

6.18.5 Initial Conditions

The atoms should have a Maxwell distribution of velocities. This is generated by making x and y separately have Gaussian distributions.

In order to see a noticable effect of atom-piston collisions, the mass of the piston was made 100 times the mass of an atom.

6.18.6 Equal Time Steps

For animation purposes, we wish to advance time forward in equal time steps, rather than in 'next-interesting-event' steps. This is easily done by starting with a frame-time step, subtracting the interesting Δt from it each interesting-step, until the next Δt is beyond the remainder of the frame-time step. Finally, updating positions for the remainder of the frame-time step.

A Δp value can be maintained over the frame-time step, accumulating it for each atom collision. This is used to display the upwards accelleraton arrow on the piston.

6.18.7 Statistical Oscillations

The motion of the piston should be statistically random. It was found that this is not the case, it seemed to oscillate at some characteristic frequency. This can be explained by considering the piston-gas combination as a resonant oscillator. The pressure of the gas produces a spring force. The two will tend to oscillate at a specific frequency. The random collisions of atoms will provide a spectrum of excitation frequencies, but all will disappear into noise but the resonant frequency. This effect is exaggerated if the P, V, T

relationship does not started out with the system in thermodynamic equilibrium. In practice, an initial volume was selected to cause the ideal gas law to be obeyed and the parameters chosen to make the oscillation be as long duration (and thus as little noticeable) as possible.

6.19 Central Force Laws

Several physical processes can be described by forces acting between pairs of particles that are functions only of the distance between them, and that act in the direction on a line between them. (Warning to the reader, I might have the sign backward in some of the force laws below).

Let us consider the arbitrary two dimensional case, where each particle is at (x_i, y_i). The motion is found by calculating the accelleration of each particle due to the sum of the forces acting on it. The contribution to the accelleration of particle j due to particle i is found by first calculating:

$$\begin{aligned} \Delta x &= x_i - x_j \\ \Delta y &= y_i - y_j \\ r &= \sqrt{\Delta x^2 + \Delta y^2} \end{aligned}$$

the force *vector* is

$$\begin{aligned} \mathbf{F} &= F(r)\hat{\mathbf{r}} \\ &= F(r)\left(\frac{\Delta x}{r}, \frac{\Delta y}{r}\right) \\ &= \frac{F(r)}{r}(\Delta x, \Delta y) \end{aligned}$$

Note the division by r. Even if $F(r)$ is inverse square you still can't avoid taking the square root. The accellerations are

$$\begin{aligned} \mathbf{a}_i &= \mathbf{F}/m_i \\ \mathbf{a}_j &= -\mathbf{F}/m_j \end{aligned}$$

Typical code to accumulate accellerations due to all combinations of N particles is

```
FOR I=1 TO N
    AX(I)=0;    AY(I)=0
FOR I=1 TO N-1
    FOR J=I+1 TO N
```

```
DX=X(I)-X(J);          DY=Y(I)-Y(J)
R=SQRT(DX*DX+DY*DY)
FORCE=F(R)
FX=FORCE*DX/R;         FY=FORCE*DY/R
AX(I)=AX(I)-FX/M(I); AY(I)=AY(I)-FX/M(I)
AX(J)=AX(J)+FX/M(J); AY(J)=AY(J)+FX/M(J)
```

The simplest possible integration technique is Euler's method. It basically consists of adding Δt times a derivative to a value. In terms of position, x, velocity, v, and accelleration, a this is:

$$\begin{aligned} x_{new} &= x_{old} + v_{old}\,\Delta t \\ v_{new} &= v_{old} + a_{old}\,\Delta t \end{aligned}$$

This technique is not very accurate. A better approach is the "leapfrog" method. This calculates velocities at time steps halfway between those of position and accelleration. This works nicely because the velocity at the halfway point is a good approximation to

$$\frac{x_{new} - x_{old}}{\Delta t} = v_{mid}$$

The position is then kept at integral time steps, and the velocity is kept at half integral time steps. If the accelleration is a function of just the positions, the update cycle looks something like:

$$\begin{aligned} x_1 &= x_0 + v_{.5}\,\Delta t \\ a_1 &= f(x_1) \\ v_{1.5} &= v_{.5} + a_1\,\Delta t \end{aligned}$$

6.20 Gravitational Force

The many body problem in gravitational fields can be easily integrated in this way with

$$F(r) = -G\frac{m_i m_j}{r^2}$$

A little time can be saved by combining the multiplication and division by masses. This also allows for zero mass objects, those which experience a gravitational force but dont generate one. The last four lines of the general purpose code above is replaced by:

```
TEMP =G /(R*R*R)
TX=TEMP*DX           ; TY=TEMP*DY
AX(I)=AX(I)-TX*M(J); AY(I)=AY(I)-TY*M(J)
AX(J)=AX(J)+TX*M(I); AY(J)=AY(J)+TY*M(I)
```

This technique was used to show

1. Three body problem showing conservation of momentum
2. Shepherding sattellites of particles in the F ring of Saturn

Any needed animations of the two body problem were solved explicitly using Kepler's equation.

6.21 Lennard-Jones Potential

Forces between atoms can be approximately modelled by the Lennard-Jones potential, defined as a function of distance from an atom:

$$U(r) = -U_0(2\left(\frac{r_0}{r}\right)^6 - \left(\frac{r_0}{r}\right)^{12})$$

where r_0 is the radius at which U is a minimum and the force between atoms is zero. The value for r_0 for a particular pair of atoms is just the sum of their individual radii $r_i + r_j$.

This leads to the force law

$$\begin{aligned} F(r) &= -\frac{dU}{dr} \\ &= 12U_0\frac{1}{r}\left(\frac{r_0}{r}\right)^6\left[\left(\frac{r_0}{r}\right)^6 - 1\right] \end{aligned}$$

The fact that this is an odd power of r is nice because it combines with the division by r in the general case to allow us to avoid the square root.

Code to calculate the accellerations looks like

```
RSQD = DX*DX + DY*DY
SIGMA = R(I)+R(J)
RATIO2 = (SIGMA*SIGMA)/RSQD
RATIO6 = RATIO2*RATIO2*RATIO2
FCOMMON = RATIO*(1.-RATIO)/RSQD
FX = DX*FCOMMON;      FY = DY*FCOMMON
AX(I)=AX(I)-FX/M(I); AY(I)=AY(I)-FX/M(I)
AX(J)=AX(J)+FX/M(J); AY(J)=AY(J)+FX/M(J)
```

Some variants were introduced into the simulation to provide for

- A constant downward gravitational force.
- Bouncing atoms off walls/floor. These can be combined to give a completely enclosed box.

- Atoms escaping through an opening in container.
- Flagging various atoms for color (blue/brown) and whether or not to show a superimposed momentum vector.
- Adding constant electric field and charge to selected atoms. This was used to schematically illustrate electrical conduction in metals. In this system, electrical resistance is caused by electrons bouncing off imperfections in the crystal. Imperfections in the crystal are really thermodynamic motions etc. Showed by brown atoms. Free electrons subject to uniform accelleration due to electric field and interact by elastic collisions only with brown atoms.

Part IV

ENGINEERING

Part IV (ENGINEERING) discusses the implementation of the conceps described so far. In it I will talk about: specific software, how various parts communicate, concrete details about getting design ideas into data files, production and delivery of the final product.

Chapter 7

Software Tools

The programs seem to fall into several categories

- Rendering Programs

  These read in some commands and databases and draw a picture on the FB.

- Design Programs

  These are

  - Digitizers,
  - Procedural Database generators

    mathematical calculations to generate list of primitives for use by rendering progs

  examples: `DIGIT, CMN`

- Animation Programs

  Have two functions

  - edit and preview scene motion on Picture System
  - write rendering data to files and invoke renderers

  can be generic: `ARTIC`
  special purpose with built in calculations: `SPACE, LJ`

Use if binary datafiles avoided.

7.1 Command Language Interpreter

Almost all programs are operated by a common command language mechanism.

7.1.1 External Appearance

To begin with, commands consist of a keyword followed by a list of parameters. Commands look like

```
XXXX par,par,par
```

Initially commands come from the terminal, but a built-in command called `READ` can redirect input to come from a file. Such file redirection can be nested.

Numeric parameters can be optionally specified symbolically. An internal symbol table is referred to in this case. Some built-in commands provide for setting and displaying entries in the table: `SET A,5` and `PRNT A`. (These are handy for some animation applications described below) Very simple expressions of the form `A+5` or `B*4.2` are also supported.

7.1.2 Internal Processing

The main program calls the interpreter to get a command, branches to code specific to that command, and goes back for the next command. The command specific code calls CLI routines to grab parameters off the command line.

The interpreter routines contain some subtle mechanisms to do useful things. Routines are defined as logical functions, return `true` or `false` depending on success/failure.

Internally programs look like

```
1  IF ( KLICOM(COMAND)) STOP
   IF (KOMIS(COMAND,'BLAH')) GOTO do_BLAH
   IF (KOMIS(COMAND,'BLEH')) GOTO do_BLEH
   CALL KOMBAD(COMAND)
   GOTO 1
```

Why like this; it looks like a table.

`KLICOM` reads VMS command line for first command in one-shot mode. Transparent to program.

`KLICOM` has built in `READ` command and stack of input files. Transparent to program.

Code to do a command looks like

```
CALL KLIFLT(0.,A,0.)
CALL KLIFLT(0.,B,0.)
process parameters A and B
GOTO 1
```

KLIFLT routine gets next text up to ',' and converts to a number. If it starts with a letter, looks it up in the symbol table.

INIT command handling

7.1.3 Command Grouping

The interpreter system contains some subtle mechanisms to greatly enhance the programmers ability to utilize various previously defined code modules and subroutine libraries and integrate them into a new main program.

The command language interpreter allows a group of commands to be interpreted by a separate set of library routines. By referencing the library, the programmer then automatically provides the user with any input commands needed to manipulate the entity that the library deals with.

For example consider a library to manipulate transformation matrices. Programmer callable routines exist for maintaining a "current transformation", a matrix stack, and for applying rotations, translations, scales, etc. to the current transformation. The sub-command interpreter routine will then accept a command keyword recieved from the CLI parser and generate a call to the appropriate routine.

The command interpretation loop of the main program then includes a call to this sub-command interpreter as well as its own commands to draw some primitive object according to the current transformation. The user of the program will generally first enter the transformation commands to define the position of the object and then enter the drawing commands.

Such topical command groupings and their associated function libraries and internal databases have, in fact become a major programming element within the system. Such a structure corresponds in many respects with the concept of a "class" in Smalltalk systems. Ultimately a main program would simply consist of a series of initialization calls, the command interpretation loop, and nothing else.

Code for inclusion of command groups looks like

```
1  IF ( KLICOM(COMAND)) STOP
   IF (KOMxxx(COMAND)) GOTO  1
   IF (KOMIS(COMAND,'BLEH')) GOTO do_BLEH
   CALL KOMBAD(COMAND)
   GOTO 1
```

The **KOMxxx** routine looks at the command, if it recognizes it it performs the command and returns TRUE. If it doesnt recognize it, it returs FALSE. It is a part of a module that deals with a specific database. This provides for separation of functionality.

Modularization of groupable functions. e.g. transformation stack
KOMxxx interprets keyboard commands
Internal accessing/inquiry commands
Information hiding, also called 'objects'.

Usage

Examples of data to be sent to line drawing program, FBDRAWS

```
PUSH
PERS 45
TRAN 0,0,10
COLR 1,1,1
MOVE 0,0,0
DRAW 1,1,0
POP
```

Can

1. enter manually
2. put all in file and READ file
3. Put databases in file, type transf and READ database
4. have one file with transf that contains READ command (These are what are going to be automatically generated by animation programs)

Program doesnt see READ nesting

7.2 Modelling

The modeling process is basically that of database generation. Such databases generally consist of text files of commands to the various rendering programs, and binary files of texture maps.

7.2.1 Geometric Modeling

The most often used modeling program in the system is the text editor. Quite a few of the modeled objects were generated essentially manually by reading and measuring blueprints or other diagrams and simply typing the coordinates into a file, along with the appropriate commands for the renderer. This approach is not as unpleasant as it may seem however. In many cases, the actual coordinates or sizes of objects are already marked on the blueprints. In addition, certain "medium level" primitive commands are provided in the pass 1 portion of the rendering program. These consist of single commands for such frequently encountered shapes as boxes, surfaces of revolution, or tubular struts. The medium level primitives are then automatically expanded into the proper set of low level primitives when the pass 1 executes. In addition, the inclusion of comments within this database file provides a valuable documentation for future modifications to the model. Finally, the use of symbolic parameters for various quantities is already built in to the command language interpreter. Parametrized models are therefore easy to create. It must be noted, however, that if this technique is used, any programs processing the model might need to keep these parameters in unexpanded form to properly process the now implicitly procedural model.

In addition to manual editing, there are a few simple menu/tablet driven modeling programs. These programs all allow more natural "drawing" modes of input to edit the location, sizes and types of object primitives. In each case the programs can read in a text file containing commands to the rendering program, build an appropriate internal data structure, allow the user to edit the structure interactively and finally write out a new command file. Such programs currently exist for patch design, polygon digitizing, and composite object design using a list of arbitrary primitive shapes. In the latter case the user is not so much designing shapes as a "tree of transformations" as discussed below.

Various special purpose design programs have also been written for various projects. Basically, given the definition of the format of the text commands for a particular rendering program is is fairly easy to quickly put together programs to generate data in that format via some desired algorithm.

7.2.2 Texture Pattern Generation

In addition to designing shapes it is also necessary to generate texture patterns for the various texture mapping programs.

Simple one dimensional textures (e.g. rings) are generated via a general

purpose curve manipulation command. This provides a means of performing various simple arithmetic operations on tables of numbers derived from previously digitized images.

Two dimensional texture patterns are derived in a variety of ways. Generally, a texture pattern is taken from a region of the frame buffer so that any image synthesis or image processing program can be used to generate or alter a pattern. We have a collection of general purpose random number drawers, image rotation and stretching programs, image filtering programs, etc. In addition, some special purpose programs have been written to process images of moons into maps. These effectively run the image synthesis process in reverse, distorting the image into a map projection.

Finally, frame buffer painting programs can be considered as a database generation tool for texture mapping. Certain special features have also been added here that are useful for spherical mapping. One interesting example concerns the generation of the terrain map for Mimas. A photograph of the moon gives a good general feel for the topography of the surface. Automatic methods for extracting this topography from the sighting parameter were not successful, however. A topographic map was finally obtained by using a painting-style program in "pantograph" mode. A map projected image of the surface was placed on the bottom half of the display and the operator traced out the visible craters. The craters were simultaneously applied to the actual map on the top of the screen by means of a special "carving" brush mode. This mode added or subtracted values from the image to raise or lower the generated terrain.

7.3 Rendering - Low Level

These read a command stream and draw a picture on FB commands are

1) Parameter setting

2) Rendering primitive under current parameter list

examples: FBDRAWS, GSET, PY123, PLANET, STARDRW, RINGS show variants of PLANET

This section will deal with various low level rendering programs. The term low level refers to the fact that they each deal with one type of object and will typically be used in sequence by the high level rendering programs or animation program to build up a composite image.

We have, in our tool kit, the standard collection of rendering programs which operate on databases consisting of polygons, quadric surfaces or patches. In addition there are several rendering programs for special cases of these shapes. For example there is a program that draws textured spheres, used for drawing planets. There is another program for drawing rings, etc.

The special purpose programs take advantage of the special properties of their specific object to increase the rendering speed or improve the visual appearance.

7.3.1 General Operation of Rendering Programs

The particular technique used for high level image composition will determine the general strategy which must be used by the low level programs. In our case the high level image composition is performed by temporal priority. To be compatible with this technique, each low lever rendering program simply overlays its image in the frame buffer on top of what is there already. Currently, all programs operate in scan line order.

In each case, the general operation of the program is the same. The operator gives commands to set various viewing parameters, surface properties, and modeling transformations. Other commands cause the appropriate primitive element to be generated. For some special case programs, a single DRAW command will initiate the rendering of the appropriate object according the the currently set viewing/shading parameters. For programs that deal with collections of more general primitives, each primitive drawing command passes the data down the pipeline to be accumulated in a buffer. The DRAW command then initiates the sorting and rendering of this buffer. For debugging/testing these commands may be entered manually on the keyboard. Alternatively, just the initial view selection comes from the keyboard and the input is redirected to a file to read a more complex model for rendering. When used with the high level image rendering programs, the view selection commands are automatically generated by another program.

7.3.2 Special Purpose Renderers

There are a wide variety of special purpose renderers in use for various purposes. For the most part they are fairly simple programs that overlay some fairly simple image in the frame buffer. Some examples are: a star field drawing program that references a large star database and draws anti-aliased dots on the screen, or a program that draws a two dimensional blurred spot to represent the sun. Some of the special purpose programs are more complex and deserve special mention.

PLANET (Draw Textured Ellipsoids)

An example special purpose rendering program is PLANET, which draws textured spheres. Since it draws only one sphere at a time it doesn't bother

with z sorting or testing. In addition to the the standard viewing and lighting specification parameters it has some other special purpose features necessary for space scenes.

The most obvious is texture mapping. The texture map is referenced while the image is scanned out to vary the surface color, normal vector perturbation, or specularity. Texture maps consist of images saved with the standard image saving utility. Thi image is typically divided in memory into "tiles". Each tile is a 32×32 square pixel sub-array of the map. This method of subdividing the map minimizes the chance of thrashing in virtual memory of the VAX.

Another special feature provides for shadowing effects of a ring structure surrounding the planet. For each pixel in the image the appropriate geometric calculations are performed to intersect the line between the planet surface and light source with the ring plane. The radius of this intersection is then used to index a one dimensional texture map containing the radial density distribution of ring particles. The brightness of the pixel is then reduced appropriate to the blocking effect of the rings.

Another special feature is eclipse simulation. This is another shadowing effect due, in this case, to another spherical body in front of the sun. In this case, for each pixel, the programs determines the angular separation between the occulting body and the sun. This is used to calculate the proportion of the sun's disk which is blocked off by the eclipsing planet and again reduces the intensity of the pixel appropriately. This has the effect of simulating the umbra/penumbra properly for eclipses. Since these two shadowing calculations slow down the rendering process they may be enabled by a global switch within the program, and are thus only used when necessary.

`RINGS` (Draw Textured Translucent Disks)

Another special purpose program is RINGS which draws an arbitrarily oriented circular translucent disk. It will, in fact draw only half the disk, where the break occurs at the plane parallel to the viewing direction which passes through the center of the disk. This split is used to properly draw a ringed planet by running RINGS to draw the back half, then PLANET to draw the disk of the planet, and RINGS again to draw the front half. The intensity and transparency of the rings are calculated using a reflection model appropriate to clouds of particles. The density and albedo of the ring particles is taken from a one dimensional texture pattern file which is indexed by the radius of the visible point from the ring center. In addition, there is included a procedural model for some radial structures in the rings which orbit the planet at different rates at different radii.

TEXFLY (Square Texture Mapper)

Another useful special purpose program is TEXFLY, which has a single textured square as its sole primitive. The square can be arbitrarily scaled and oriented in three dimensions and rendered with various intensity and transparency patterns. The very simple geometry of this shape makes it run quite quickly. TEXFLY is used, for example, to generate schematic explosions by scaling up and fading out a painted explosion pattern.

Miscellany

Other special purpose renderers were built, usually cannibalizing parts of existing general purpose renderers, for the following purposes:

`FBFIELD` version of **FBDRAWS** that integrates field lines and sends them down rendering pipe

`RIPPLE` makes black/white ripples for light wave interference patterns

`QM3D` displays probability densities of Hydrogen orbitals

`PVT` Version of **TEXFLY** with built in texture pattern to draw phase diagram from van der Waals equation.

7.3.3 General Purpose Renderers

FBDRAWS (Draw 3D colored lines and polygons)

This was the renderer that was used most often in the animations. It simply makes 3D colored line drawings with all the normal transformation and clipping operations applied. As the project progressed more and more features were added so that it is now a fairly complex program. Extra features include:

- Sorting of line segments according to z depth. The lines are then rendered in back to front order.
- Variable thickness lines. Also the intensity contour across a line is variable, allowing darker edges. This gives a subtle cylinder-like quality.
- Extra clipping planes can be specified in addition to *top, bottom, left, right, near* and *far*. This was used primarily to split objects into pieces so that the temporal priority style of object merging would work.

- Filled polygon primitives. A polygon can be optionally drawn based on whether its projected area is positive or negative. This is a back-facing polygon cull. Polygons are not shaded though, they are just one solid color.

- A character generator that can be programmed with databases from the Hershey character set. This was used sparingly. It usually proved best to make character strings into explicit wire frame line primitives because the animation program could display them properly. The built-in character generator of `FBDRAWS` was used for digital-display of numeric information.

- A mesh generator for making altitude plots of functions of two variables. This just made it easier to specify a set of strips along the xy mesh.

In addition, hooks are provided so that it is easy to make special versions with procedurally defined wireframe primitives. An example of this is the field line integrator program `FBFIELD` mentioned above.

The Three Pass Process

More general programs which operate on collections of primitives (polygons, patches, etc.) in scan line order are usually separated into three passes.

1. Interpret viewing/modeling transformations, assigns shading parameters, and transforms the primitive objects and accumulates them in a buffer.

2. Sort this buffer in y order.

3. Render the image from the y sorted list.

To maximize available buffer size—and thus the complexity of objects we can draw—these three passes are implemented as separate programs. The y sorted buffer is implemented as a temporary file. The I/O operations on this temporary file do slow down the process a bit, but not a great deal. The advantage gained is that the rendering pass only needs maintain in main memory only those elements which are active on the current scan line.

Three sets of programs have currently been implemented using this strategy. They deal with polygons, bicubic patches, and blobbies. The databases to describe shapes to these programs are then simply command files which can be generated in a variety of ways.

In addition, for speed, pass 1 of the polygon processor can also interpret a binary version of a command file. A simple pre-processor reads the text command file and generates the binary file. Thus objects which are going to be drawn repeatedly may be debugged via the text version of the model and then "compiled" into the binary version.

Special Purpose Pass 1 Processors

The use of temporary files for the y list allows another degree of flexibility. For certain cases the geometry of a particular object allows a simpler internal representation than explicitly defining each individual primitive. In this case a special program can use this representation to generate the y list file directly in sorted order. This would then replace the general purpose pass 1 program and eliminate the need to run the pass 2 sorter.

One example of this technique was used in the DNA simulation. In this case, the large macro-molecule is made of a relatively few monomers of 25 to 30 atoms each. A large storage and time reduction can be achieved by representing the molecule in terms of the locations and orientations of these monomers and storing the definition of a monomer only once. The y list of the individual atoms may be generated directly by sorting the monomers first. This list is then scanned in order of their y appearance, expanding the monomers into the explicit atom list.

Another example concerns a simple terrain rendering scheme. The altitudes of a region of terrain are generated assuming a regular grid spacing. These can be stored in an appropriately scaled byte array. Pointers into this array are then sorted in y. A global y scan then directly generates a y sorted polygon list by assuming the connectivity of the points due to the regular grid spacing. This technique was used to simulate a fly-over of a crater on the moon Mimas for the second Voyager Saturn encounter movie.

7.4 Rendering - High Level

High level rendering techniques concern themselves with building up images consisting of several disparate low level primitive types. The idea here is that the low level rendering programs do not need to know anything, or at least very little, about each others' operation. The high level rendering technique causes them to be invoked in the proper manner.

7.4.1 Techniques

The main problem the high level rendering system must solve is the occlusion problem. There are several strategies which may be employed, of

which the two most popular are Z buffers and Temporal priority (also called the Painter's algorithm).

Z Buffers

With Z buffers, the low level primitives must all reference a common Z buffer and will presumably have their Z values scaled into the same coordinate space. Otherwise they do not need to know about each other. Using this scheme, the high level scheduler can invoke the low level programs in virtually any order. This is convenient for complex texture mapping since all objects painted with a particular pattern can be rendered at once. Thus only one texture needs to be referenced at a time. Z buffers also solve the problem of arbitrarily intersecting primitives easily.

Z buffers, however, have some severe problems for the space images considered here. For space scenes, the very large range of values in Z would require a Z buffer with very many bits of precision to have the resolution necessary to keep things properly hidden. In addition, anti-aliasing calculations are inconvenient with this scheme.

Temporal Priority

The basic idea behind temporal priority is to draw the objects in the scene in the order back to front. The later objects simply overlay the earlier objects. It is the technique used for almost all compositing in the scenes in MU. Temporal priority works well for space scenes because the different objects do not intersect (or at least are not supposed to). It does not work well if objects interpenetrate or if there is cyclic overlap in portions of two objects. In this case objects must be split into parts that do not intersect or overlap. For some scenes this was a monumental pain.

7.4.2 Space Simulation Global Scheduler

The high level rendering scheduler for space scenes is a fairly specialized program which takes the frame state information file from the space simulator program and generates the command files to render the image on the frame buffer. It consists of four phases:

Global clipping An enclosing sphere about each object (moon or spacecraft) is tested against the viewing volume. Those objects completely outside this volume are removed from further consideration.

Global Z sort This then sorts the remaining probably visible objects in back to front order.

Generation of individual command files A command file is generated for each visible object consisting of the transformation commands for placing it in viewing space, various surface lighting and texture map definition commands, etc. At this point some mutual shadowing calculations are performed. If a moon is in shadow behind the planet its brightness parameter, written out here, is decreased by 1/4.

Generation of global system command file This is a system command file that runs the appropriate rendering programs in the Z sorted order and tells them to read their commands from the appropriate file written by the previous step. The appropriate rendering program to use for a particular body is taken from a table which is set up upon initialization of the high order renderer. Here also some shadowing calculations are performed. If a line from the sun to a moon intersects the planet on the sunlit side an extra command is added to the global file to run a program that places a black dot at the appropriate screen location.

Finally, control is given to the system to read commands, not from the file in step 4 but from another static file. This file in turn references the step 4 file and then does some clean-up operations, such as saving the image. The overhead for all this file manipulation may seem excessive but, in practice, it is negligible compared to the running time of the primitive renderers.

There are several useful features to this approach. First it allows for easy retry or debugging of an image since the commands and data files remain after the program terminates. Secondly it allows for some after-the-fact changes to the system command file by inserting suitable commands before or after its invocation by the global file. Two cases where this was used were during the title sequence of the Voyager Saturn movie. In these cases an explicit run of `TEXFLY` for a record overlay, or the polygon renderer, for the title letters, was inserted just after the invocation of the step 4 file.

7.5 Animation

The main function of an animation program is to alter, as a function of time, various numerical parameters that define the appearance of the frame. A subsidiary, but very important, function is to provide a schematic preview of the animation on some fast output device. This is typically a line drawing display, the preview is effectively a "line test" of the animation.

7.5.1 Levels of animation software

A categorization of levels of sophistication that is becoming popular is

Keyframe Explicit values for each parameter are listed and associated with frame numbers.

Simulation Various physical constraints or dynamic processes are applied to automatically generate all or most of the parameter values.

Psychology A set of goals or objectives are specified and some artificial intellegence program attempts to meet them.

The animation systems I have devised only use the first two of these techniques. The keyframe system, while it requires the animator to specify a lot of information also *allows* the animator to specify motion. Sometimes high level systems produce strange results. It is important to be able to edit or override such unwanted effects by explicit low level controls where necessary.

We will devote an entire section to one special one, `ARTIC`.

7.5.2 Techniques

There are two general categories of parameters which change from frame to frame. They are 1) transformation parameters and 2) anything else. Transformation parameters are singled out especially here because they are so important. Typically a scene will be defined in terms of a "tree of transformations". This consists of the set of nested rotations, translations and scales that place each object in its correct position at its correct size. The nesting of these transformations allows objects to be arbitrarily articulated. The numerical values assigned to these transformations can then be varied to give a wide variety of motions. Many animations require no more than this. Other more general types require, in addition, alteration of brightnesses, texture patterns etc.

There are, in turn, two general ways of specifying the way parameters vary, called here incremental and absolute. In the absolute mode, the value of each parameter is specified in a table for each key frame. When an intermediate frame is to be generated, some form of interpolation is performed on the surrounding table values. In the incremental mode, the program maintains the current state of the scene and is instructed to perform incremental modifications to it according to some rules. Absolute mode is usually more convenient to use when designing animation since it allows previewing of frames in arbitrary order or repeated playbacks of frame number ranges within the script. Incremental mode is necessary, however, for situations in which the connectivity of some data structure must be modified or in "particle pushing" types of physical simulations.

7.5.3 Tools

System Commands

One animation tool that everyone with a computer has immediately is the command language of the operating system. Most operating systems allow for commands to come from files and, in addition, allow for looping and symbolic variable manipulation within the command file. This mechanism has a tremendous advantage in that it is interpretive. Changes to the sequence of events necessary to make an image can therefore be made quickly with the text editor. No programs need to be changed. A disadvantage of using just the system command language is that it is quite difficult to perform line tests to verify that the animation will come out as desired. Also, since the command language was not designed for this purpose, its use sometimes becomes somewhat inelegant. The advantages in generality provided by the interpretive nature are substantial however and the method described below allows its use in combination with more specialized simulation programs.

Special Purpose Space Simulator

For various applications the values of the transformation parameters may need to be generated by some functional calculation rather than by explicit numeric settings. An example of this is the space simulation program used for the Voyager fly-by movies. This is a special purpose program for simulating planetary astronomy. Its two main features are the mathematical modelling of the paths of the planets, moons and spacecraft according to Kepler's laws and a fairly sophisticated view selection algorithm for determining viewing position and direction in terms of some visually meaningful parameters.

Physical Simulation The main structure of the modeled environment consists of a built-in transformation tree for the planetary system. This positions the moons and spacecraft relative to the planet, orients the planet and moons with their pole vectors pointing in the appropriate directions and spins them about their poles at the correct rates. In addition the spacecraft is oriented and articulated in an analog of the manner done by the onboard computer of the spacecraft. The values of positions, and rotation angles are made dependant upon the single parameter *time*. Whenever time is altered the appropriate calculations are performed to update these values.

View Selection Since the positions and orientations of the objects are completely specified by the time of the simulation, the only freedom we

have in specification of the image is the viewing location and direction. There are several viewing modes which may be employed to generate these two parameters.

Omnsscient Mode The main viewing mode is for an omniscient observer situated at some vector offset from any one of the simulated bodies, called the 'from' body. This vector may be specified in one of two ways, as a fixed x, y, z vector or as a fixed distance and an automatically calculated direction. The latter mode determines what direction relative to the 'from' body the observer must be in in order for the 'from' body to appear on the screen at a given x, y location. The viewing direction is also determined according to one of two modes. It may be explicitly specified in terms of its x, y, z coordinates. Alternatively it may be specified in terms of another simulated body, the 'at' body. In this mode the viewing direction is automatically calculated to be that which causes the 'at' body to appear at a given x, y screen location. For certain combinations of from/at modes there is no closed form equation to satisfy all the constraints. A solution is then determined numerically.

Camera Mode An alternative viewing mode is to simulate a view through the onboard cameras of the spacecraft. In this case the spacecraft modeling parameters are examined and the appropriate view is generated according to the pointing direction of the camera and the spacecraft orientation.

Planet Surface Mode This mode simulates a viewer sitting on the surface of one of the simulated bodies. The viewing position is specified in terms of the latitude, longitude and altitude above the surface. The viewing direction is specified in terms of the azimuth and elevation of the view direction relative to the local horizon and north direction.

Switching Viewing Modes The various viewing modes provide a great deal of flexibility in finding interesting views in a given situation. One important feature is also necessary for smooth animation. Whenever the user switches modes the program will automatically calculate the appropriate numerical parameters for the new mode which will generate the same view. This allows easy transitions from one mode to another without introducing jumps in the picture.

Animation Commands The animation of space scenes is carried out in absolute mode. There is a table of the values of the various viewing

parameters as well as a few spacecraft articulation parameters, for each key frame. A new key frame is usually generated by adjusting various parameters under knob control. A command is then given to record the current parameters in the animation keyframe table at a certain keyframe number. When an animated sequence is played back, all the table values are interpolated appropriately and given back to the view generation routines.

When the space simulation program is instructed to make a color frame it writes out the values of all the internal parameters and invokes the frame buffer scene scheduler program described in section ??.

Special Purpose DNA simulation

The DNA simulation system is another special purpose articulation program which has many built-in parameter calculations and transformation tree structures. In this case, the physical simulation is quite complex while the view selection is quite simple.

View Selection The interpolation of viewing parameters is done in a manner similar to the space simulation, but with a considerably simpler method of specifying viewing parameters. The view is specified simply by a field of view, center of interest point, distance from view point angular direction of observer and tilt of camera. These parameters may be adjusted via knob control. When a desired view was found they were stored in a keyframe table which was used for interpolation upon playback.

Physical Simulation The DNA simulation program represents the molecular system as a collection of modules which are each rigid bodies and are connected together at rotatable joints. Each monomer is defined in terms of the location and orientation of its joints and as a list of the locations of its constituent atoms. Whenever a particular monomer is moved or rotated, a recursive connection tracer applies the same transformation to all other modules bonded to it. Each connected structure is assigned a velocity and tumbling speed which, upon each frame time simulation, are added to the current position and applied to the current orientation, respectively.

Since the animation of the molecular motion consisted of breaking and relinking bonds repeatedly, the absolute mode of simulation is not appropriate. Instead, the animation is driven by commands that

- Set the velocities
- Break and re-form bonds

- Run the simulation forward by some number of time steps. The "script" for a sequence is, then, a list of such commands. First, commands to set some velocities, and break and reform bonds. Then a simulate command. Then more velocity/bond alterations. Then another simulate command, etc. The animation is then performed as the file is read. This makes moving directly to any given frame a bit difficult. If the desired frame is later than the current one, the file just continues to be read. If the desired frame is earlier than the current one, the program must be reinitialized from the beginning frame and run forward to the desired frame.

  In fact, doing the entire film by this technique would be very difficult due to the complexity of the motion. Since much of the motion is actually initiated by the two enzymes in the film, two enzyme simulation routines were added to the program. Each of these routines consisted of a finite state machine which counts simulation time steps, performs some velocity alteration and/or bond relinking operation, resets the time step counter to a new value, and then changes to a new state. For example, the helicase simulation has two states

  1. Prying apart
  2. Moving down

  At each time step, the first state rotates the two separate strands about their bonds by a small incremental angle and rotates the main double helix in the reverse direction by the same amount. The second state moves the helicase down one base pair.

  The polymerase simulation has four states

  1. Waiting for a nucleotide
  2. Pushing the nucleotide in place
  3. Retracting and moving to the next nucleotide position
  4. Removing an erroneously matched nucleotide

  The transition from state 3 to state 1, for example, generates a new incoming nucleotide by adding it to the database in such a position and at such a speed that it would fly into place just as state 1 expires and changes to state 2.

  The randomly floating background nucleotides were driven by a simple program that generates their paths randomly, but in such a manner that they do not collide with the main DNA strand. The output

of this program, when interspersed with enzyme directing commands then forms the main script of the movie.

When the simulator program is instructed to make a color frame, it writes the locations and orientations of the modules to a temporary file and invokes the module expansion program described in section ???.

7.6 ARTIC

Almost all animations were done with a general purpose articulation program called ARTIC. This is the largest and most elaborate program I have ever written. In doing so, I had to be very careful to avoid absolute anarchy in the internal design. Various features were added only if they were *really* needed.

7.6.1 Desired Functionality

Example

Let us begin with a concrete example. Consider the following list of commands that are to be sent to a rendering program.

```
PUSH          "Push current transform"
PERS  45      "Perspective field of view=45"
TRAN  0,0,10  "Translate back 10 in Z"
ROT   30,2    "Rotate 30 degrees about Y"
COLR  1,1,1   "Set color to yellow"
READ  CUBE    "Read CUBE database"
POP           "Pop current transform"
```

Individual frames might be animated by altering the values of the parameters to the TRAN and ROT commands. Sending each such altered list will make a different frame of the movie.

To make it easy to see what will happen we would like a program that will do the following

- Keep a list of transformation and drawing commands
- Allow alterations to the list
- Rapidly interprets the list to show results on a line drawing device such as a Picture System
- Writes the edited list to temporary file to be fed to a rendering program

Conclusions

There are two main functions of an animation program; it is a Previewer and a Macro Expander. The Preview function refers to the

design and playback of motion on some fast line-drawing device. The Macro Expansion function refers to generation of command files to feed to rendering programs to make the final images.

There are a number of animation systems that I have written or seen that just do the macro expansion process, (I'm sure you can write one on UNIX in 10 minutes using *yacc* or *awk* or *bletch* or some such) or that just do the previewing process. Many just act as filter or compiler of some motion description language — no previewing or adjusting is provided. But you spend most of your time adjusting. My interest here is in combining these two functions.

ARTIC is not tied to a particular type of rendering program. It's primitive shapes are simply files containing Picture Systrem commands (referred to henceforth as .PS files), or as text files containing 3D MOVE and DRAW commands (referred to henceforth as .LIN files). When sent to a renderer via the macro-expansion phase, just the first part of the file name is sent. The renderer then reads in another file describing the object in the language the renderer needs.

This requires a set of utility programs to convert various renderer-oriented primitive databases to .LIN or .PS files.

ARTIC could be used do generate command files for a renderer that has no line drawing representation, but its real dangerous to animate something without some preview.

7.6.2 Basic Token Processor

The best way to think of ARTIC is similar to a text editor. It then has commands in the following categories.

Read old database

Read in a text version of a list of transformation and drawing commands. These are tokenized into an internal data structure that provides for rapid interpretation. Such files are called .SKLfiles (the transformation tree is a *skeleton*.

Primitives are defined by reading in a .PS or .LIN file. Names of such files are saved in a symbol table for later use in rendering.

Primitives are referenced by a DRAW command in the the .SKLfile.

Edit database

Allow alterations to the token list. Alterations to the list can be implemented in many ways. The way which seemed to work best is to use a spreadsheet metaphor. The token list is displayed on a text screen and the user can move around in it with cursor keys. Typing in a new value in a parameter cel will change that parameter's value.

View database

The token list can be viewed in several ways. The edit mode provides one way, as a text listing of the commands and their parameters. Another obvious way is to have an interpreter program execute the commands to draw the image on the Picture System. Each `DRAW` token in the database refers to a list of Picture system instructions which are rendered according to the current transformation matrix.

Render database

A separate command initiates the process of sending the token list to a rendering program. The keyboard command for this is `DRAW`; an unfortunate double usage of this command depending on context.

Anyway, this command de-tokenizes the database and writes out a temporary text file (`.TMP`) containing all commands and their parameters. Some slight editing is done to change "`DRAW` *primname*" in the token list into "`READ` *primname*`.EXT`" in the temp file. The strings for '`READ`' and '`.EXT`' are global parameters that can be changed to accommodate the expected input requirements of various rendering programs.

Then the rendering process invokes the operating system to execute commands from a file called `xx.COM`. Again the string "xx" can be set by an `ARTIC` command.

`xx.COM` is simply any VMS operating system commands. It typically will be set up to contain commands of the sort:

```
$CLR                    !clear screen
$RENDERER READ xx.TMP   !run program to read .TMP file
$save picture           !save picture
```

This rather involved process is very flexible. It can be thought of as a patch panel approach to program communication. Many levels of the

operation are driven by interpreters whose command lists are simple text files.

Saving database

Just like a text editor, when ARTIC exits it will write out another version of the .SKLfile if any changes have been made to the token list. Saving only happens if there are any changes made. It is important to try to prevent the user from exiting without saving changes.

7.6.3 Transparent commands

Renderers always have commands or parameters that were not thought of when ARTIC was written, or which have no obvious representation in the line-preview mode. It is therefore useful to allow arbitrary commands to be "quoted" in the token list, and fed verbatim to the rendering pass. The PS viewing routine ignores these commands. The rendering routing writes them out (almost) verbatim to the .TMP file. Two such mechanisms are provided

SET command

This looks, in the .SKLfile, like

```
SET  A,5
```

The 'A' is a one letter parameter that is blindly echoed. The '5' is numeric parameter that can be edited in screen mode. This allows setting symbolic parameters in the renderer's symbol table. A renderer's primitive object description could contain things like

```
COLR A,0,0
MOVE ...
DRAW ...
```

This works but is clumsy. More useful is.

.... command

A more general solution is a comand that looks like

```
.... XXXX,p1,p2,p3
```

The four arbitrary characters will be written out at the *beginning* of the line in the `.TMP` file, forming a command for the renderer. Exactly three numeric parameters which can be edited in screen mode just like e.g., `TRAN` parameters. This is designed to work with the syntax of our CLI.

7.6.4 Subassemblies

The next useful thing to have is subassemblies. These are effectively subroutines to the token processor. A `.SKL`file defining subassemblies looks like

```
DEF xx
...
----

DEF WORLD
...
DRAW xx
...
----
```

The Dictionary

When read in, the token list is broken into named blocks or subassemblies. Subassembly names and Primitive names are kept in the same dictionary. Whether a name is one or the other depends on how it was defined, with DEF or with PRIM. A dictionary entry has a *flag* saying if it's a primitive or a subassembly and a *pointer* to the beginning of a token list (if subassembly) or to a data structure representing a primitive (if prim). This allows the 'calling' token to be unaware of whether the called item is a primitive or subassembly; it can just use it.

Viewing and Rendering

The **viewing** algorithm threads itself through a subroutine call 'tree' starting at the built in name WORLD executing PS commands to carry out tokens. The **rendering** algorithm threads through the tree similarly but writes a text version of the tokens into a file. When either encounters a DRAW token it either "calls" the subassembly (if that's what it is) or draws the primitive (if that's what it is). The .TMP file that a renderer sees has all subassembly calls 'flattened out'.

The name WORLD can be changed via viewport setting commands. In fact, several viewports can be set up, each rendering from a different 'root' assembly name.

Other ways to do it

Another popular mechanism for accomplishing this is to have subassembly calls with prefabricated transformation matrices associated with the call. The call itself is named. Transformations are edited by referring to the call name. If you want transformations in different order, you can insert dummy levels but then you have a lot of unneeded transformations.

7.6.5 Review

What we have so far. We have a Language with

- Tokenizer of lists of instructions
- Screen editor
- Interpreter to draw on PS
- Interpreter to dump in text file
- Detokenizer for resaveing into .SKLfle

Although I have tried hard not to think of the command lists as a language, I guess is really is one.

I would like to re-iterate the importance of tokenizing a language and re-executing it *quickly* for design purposes. Most of the time is spent in diddling with parameters and transformations.

Another way to think of ARTIC. It's both

An editor using screen mode, re-saving alterations in .SKLfile

A filter transforming .SKLfiles to .TMP files

7.6.6 Levels

Why

Consider the following situation. Given the token list

```
PUSH
 PERS 45
 PUSH
  TRAN 0,3,11
  DRAW A
 POP
 PUSH
  TRAN 3,9,20
  DRAW B
 POP
POP
```

This will be fed to e.g., **FBDRAWS** which will perform the transforms, and refer to files **A.LIN** and **B.LIN** to render it.

But what if you want to use two different renderers to draw A and B? Of force B to be drawn first with A on top of it? The solution to these problems is LEVELS.

What

Instead of writing one .TMP file and calling one renderer, you can write several .TMP files and call a different renderer for each. The two files would contain

```
L1.TMP                L2.TMP

 PUSH                  PUSH
  PERS 45              PERS 45
  PUSH                 PUSH
   TRAN 0,3,11          TRAN 3,9,20
   DRAW A               DRAW B
   POP                  POP
  POP                  POP
```

Then the scene **.COM** file could be set up to contain

```
$CLR
$TEXFLY READ L1.TMP
$FBDRAWS READ L2.TMP
$save
```

How

How does it know what parts of the transform list to put into L1 or L2? It doesn't. So what has to happen is this. The .SKLinterpreter is called twice, in this case, each time with a value of 1 or 2 as a 'desired level' parameter. As it threads through the token list, it maintains a currently-active-level value. This is changed whenever it hits a LEVL token.

All the tokens are unconditionally written out except for the three tokens

```
DRAW xx
.... xxxx,1,2,3
SET  x,val
```

These are only written if the currently-active-level value matches the 'desired level' parameter.

So actual contents of temp files are

```
L1.TMP              L2.TMP

 PUSH                PUSH
  PERS 45             PERS 45
  PUSH                PUSH
   TRAN 0,3,11         TRAN 0,3,11
   DRAW A              POP
  POP                  PUSH
  PUSH                 TRAN 3,9,20
   TRAN 3,9,20         DRAW B
  POP                  POP
 POP                 POP
```

Each .TMP file will sometimes have some spurious commands, but they wont hurt anything. The '....' and SET commands need to be made sensitive to the correct level because they may represent commands that are only defined for a particular rendering program.

This technique is guaranteed to work even if levels are on/ off in strange nesting, such as

```
LEVL 1
DRAW A
PUSH
 LEVL 2
 DRAW B
 LEVL 1
 DRAW C
POP
```

Another way

Is there a way to avoid this extra garbage in .TMP files?

Another way of implementing levels may be to keep a bit mask for all active levels and be able to turn on/off any permutation of them in a LEVL command. Then output sensitive tokes only if the current-bit-mask contains the bit for the desired-level. The token list might look like

```
LEVL 1,2
stuff     "written for both L1 L2"
LEVL  1
stuff     "written only for L1 call"
LEVL  2
stuff     "written only for L2 call"
```

This could write out the cleaner versiton of L1.TMP and L2.TMP shown in the first example above. I dont know what other subtle side effects this might have though.

Who

Note that the user is responsible for costructing a .COM file that executes the renderers in the correct order, and has them read the appropriate .TMP files.

The DRAW command within `ARTIC` must call the write-out interpreter for each level. There is a table within `ARTIC` telling it how to do this. Levels are 'declared' to `ARTIC` via some keyboard commands. These place entries in the table. Entries are, for each level

1. text to convert DRAW tokens into, e.g. 'READ'
2. text to append to primitive name in DRAW command, e.g., '.LIN'
3. file name to write .TMP file to, e.g., 'L1.TMP'

Tricks

The LEVL feature allows us to do some interesting and tricky things.

LEVL 0 is useful for putting markers, labels, construction lines etc. into the wire frame view for design/previewing purposes, but not have them make their way to any of the renderers.

Another trick allows us to render backgrounds only when necessary. For example, if there's a complex background for a scene that doesn't move it can be placed in LEVL 1 and the foreground objects in LEVL 2 (even if they are rendered by the same renderer). Then the .COM file can be temporarily modified to only execute the renderer for LEVL 1. The background is then saved on a file. The .COM file is then modified so that the LEVL 1 command is replaced with a picture-restore command for the saved file and the execution of LEVL 2 reinstated.

Since the system's .COM interpreter has conditionals, this might not require changing the file. A prefix string is passed to the .COM interpreter and can be used to trigger this. The .COM file would look something like

```
$CLR
$IF 'P1'="BKGND" THEN  GOTO BKGND
$!                       Read background, draw forground
$LFF REST BKGND
$FBDRAWS READ L2.TMP
$SAVE
$EXIT
$!                       Draw and save background
$BKGND:
$TEXFLY READ L1.TMP
$LFF SAVE BKGND
$EXIT
```

This can be made arbitrarily complex. On some scenes I figured out several frame number ranges where the background remained static,

and pre-calculated and saved these backgrounds. Then I set up the .COM file to detect frame number ranges and use the correct background. This is an example of how .COM being an interpreter is useful, but perhaps dangerous if it encourages you to do such stuff.

Animation

Looking ahead to animation, note that the parameter to the LEVL command can be animated. This allows an object to be drawn on different levels at different keyframes. An example of the usefulness is in combining a polygonal/shaded object with some overlayed line labels. If the labels are behind the polygon object they should be be drawn first, if they are in front they should be drawn second. If the scene rotates, this decision changes with frame number. We use 3 levels and a symbolic value for the LEVL of the lines.

```
LEVL LLINES
DRAW LINES
LEVL 2
DRAW POLYS
```

and the .COM file has

```
$FBDRAWS READ L1.TMP  !line drawing behind polygons
$PY1     READ L2.TMP  !polygon generating pass
$PY2                  !y sort
$PY3                  !render into fb
$FBDRAWS READ L3.TMP  !line drawing in front of polygons
```

Then animate the symbol LLINES to have values of 1 or 3 during the appropriate frame number range. Some animation systems perform this sort of sorting automatically, but I'm not sure its possible in all cases. This explicit technique allows (forces ?) the user to do what he wants.

7.6.7 OPAC command

One very useful global parameter is *opacity*, called by some *alpha*. A token was added to `ARTIC` to manipulate a global OPAC value.

operation

While interpreting the token list a running value of a global opacity is kept in much the same way as the current transformation matrix. Only multiplicative modifications to this are possible (it operates like a scale factor). It is also pushed and popped by the PUSH and POP tokens. Note the effect of OPAC commands in the following:

```
DEF WORLD
DRAW CUBE      "OPAC=1 for this"
OPAC .5
PUSH
 TRAN .6,5,44
 OPAC .5
 DRAW CUBE     "OPAC=.25 for this"
POP
DRAW CUBE      "OPAC=.5 for this"
```

Use

OPAC is useful to fade things on and off. It also is useful to get rid of things that are only on the screen part time, or to swap between two versions of a database. Older systems used to do this by a common kluge I have heard lots of people use, that of transforming unwanted pieces large distances off the screen. Now we can do it by animating opacities separately for two options to switch between (0 and 1) and (1 and 0).

```
PUSH
OPAC 1    <- change to 0 or 1 to turn X on/off
DRAW X
POP
```

How renderers see it

Incidental point. Renderers have an OPAC command to use this, but currently they don't push or pop it. An OPAC command for a renderer just jams the parameter into its OPAC value. So the file-writing interpreter in ARTIC does two things when it sees an OPAC token. It updates its internal running OPAC value, and it writes out an OPAC command with the current net value. It also writes out an

OPAC command when it interprets a POP command. This is slightly messy, but doesn't worry me a lot.

Skipping Calls

Since you can only cumulatively multiply opacity values, once the opacity reaches zero it can never increase again (until popped). Therefore some economy is obtained by ignoring any DRAW tokens encountered when opacity is zero.

Skipping DRAW commands helps remove clutter but leads to some other bad drawbacks. What if some transparent-command, like a line thickness command, is included in the skipped section. Changing the opacity of one object can clobber the line thickness of another. You can avoid this by being careful of how you generate databases, but it requires constant vigilance. Or you could change `ARTIC` to not skip DRAW commands. This makes things slower and more cluttered. I'm not sure if there is a general purpose way of guarding against this problem because it comes from transparent commands, which `ARTIC` is not supposed to interpret, but just pass on.

7.6.8 Symbolic parameters

We are now ready for symbolic parameters and simple expressions. How can we attach names to angles and positions? One way is to name tokens with labels; I didn't like this.

Another symbol table

Another way: allow symbol names wherever numbers can be found. We maintain yet another symbol table, this time for floating point numbers.

example

```
PUSH
TRAN 1,Y,0
DRAW SQUARE
POP
```

Now, outside of screen edit mode, you can give `ARTIC` commands

```
SET Y,2
SET Y,5
```

After each command it redraws the PS screen, so the object moves.

Adjusting symbols

The command `ADJ Y` goes into loop looking at a knob, updating `Y` in the symbol table, and redrawing on PS. You can attach up to 8 variables to the 8 different knobs and 2 more to the x and y coordinates of the tablet in `ADJ` mode. Each knob/symbol pair has a scale factor which tells how much to change the symbol for one turn of the knob. (animation!)

Linking

Can have more than one command refer to the same symbol

```
DRAW SQUARE, TRAN, X, 0, 0,
DRAW CUBE,   TRAN, X, 1, 0,
```

Expressions

You can use simple expressions in any numeric field. These have the form

```
symbol*constant+constant
```

example of use to do simple constraint satisfying.

```
DEF  CART
DRAW WHEEL, TRAN, 1, -1, 0, ROT, A*57.1, 3
DRAW WHEEL, TRAN,-1, -1, 0, ROT, A*57.1, 3
DRAW SQUARE
----

DRAW CART  , TRAN, A, 0, 0,
```

Moving parameter A moves the cart, and rotates the wheels by the proper angle.

And then, and then...
You could allow arbitrary expressions instead of just S*a+b. The possibilities for complexity are endless. I don't do it this way because

1. Must be able to tokenize (compile) and detokenize anything you allow in token list. Although FORTRAN is the worlds most wonderful language, I will admit that recursion (very useful in compiling parenthesized expressions) is not one of its strong points.
2. Interpreting fixed S*a+b expressions is very fast, in fact, all parameters have exactly this form with perhaps a=0 for constants or b=0 for pure parameters. Parameters are then stored in a fixed table with three fields: a, b, and a pointer to symbol table entry for S. This expression accounts for 99% of all parameters anyway.
3. Long expressions don't fit nicely into fixed sized cels in screen editor mode.

Note, the interpreter evaluates the symbolic-parameter expressions, the output routine (PS or Writing) sees only floating point constants. The detokenize (Save) routine still dumps into .SKL files in expression form.

7.6.9 Review

now what do we have ...

databases

1. A token list broken into subassemblies
2. A list of PS commands for primitives, broken into separate primitives.
3. A dictionary of names for the above two.
4. A symbol table of floating point values.
5. A table of Levels and textual information on how to generate rendering commands for them.

Typical usage

The user defines a scene with token commands. Parameters can be constants or simple variable expressions. You use symbolic variables where changes are likely to be made. This makes a parametrized model.

Access to symbol table

Many parts of the software, described below, access the numeric symbol table. It is a central place for communication within ARTIC. Software access routines consist of calls to:

1. Add a new symbol
2. Look up a symbol and get its *index*
3. set the value at given *index*
4. get the value at given *index*

7.6.10 Animation tables

Now that we have symbols, we can **animate** by having symbol table entries vary as a function of frame number.

Simplest way: copy of symbol table contents at each key frame.

To play frame n; interpolate between two keyframe tables, fill in values in actual symbol table, call PS draw routine.

Playback modes

`PLAY 1000,3000,4` Loops thru, interpolating 1000,1004,...,3000. Note that it takes as long as it takes to generate frames.

`REAL 1000,3000,30` Loops thru 1000...3000 with frame skips chosen to keep global timing approximating real time. Looks at system clock each frame to see what frame number it's supposed to be at and shows that frame.

`JOG` Like 'jog' mode on a video tape machine. Attaches frame number to a knob and increases/decreases it as you turn knob clockwise-/counterclockwise. Useful for fiddling with small regions of script. Like conventional animator flipping pages of drawings.

Individual symbolic control

But..., all symbolic values might not change on each keyframe, or at same keyframes. Some might want to change on disjoint but overlapping keyframe ranges.

The keyframe table So it's better to have a set of keyframe/value pairs for each symbol table entry. For a symbol table containing

```
A  5
B  10
C  8
```

A movie table might contain

```
A:  200  4.0       B: 150  2.0      C: no entries
    300  8.2          275  3.0
    600  0            325  5.0
```

For each frame the update algorithm is: Go through the movie table, if there are key/value pairs for a particular symbol interpolate for current frame number. Call on the symbol table SET routine to put result into symbol table. Those symbols with no entries remain unchanged. This effectively separates symbols into two categories: varying and constant. In the above situation, you can `SET C,5` and replay, and only variables A and Bs get updated.

Interpretation Keyframe routines dont know or care what interpretation will be placed on symbol table entries. They just worry about interpolating and setting. This is an example of Separation of Functions, the movie routines just animate the symbol table as a function of frame number, with no other interpretation on what symbols are good for. The token interpreter then interprets symbols. This may be a disadvantage if motion blur is wanted.

Entering data The lowest level of control is a keyboard command to add a frame number/value pair to the movie table.

```
KEYF A,1000,2.6
```

Even though we have developed sophisticated commands for editing the keyframe table, this command is still available. It is the way to enter stuff generated by other programs. It is also the way the movie table is saved/restored. These commands are written out to a `.MOV` file when `ARTIC` exits (if the table has changed since the last read-in). The keyframe table is rebuilt when `ARTIC` is run later by using the READ command.

Managing the complexity

Now we have a problem. A large database of key/value pairs that the user needs to specify. It could get complex, we want to manipulate and view easily.

Various techniques were experimented with. Some commands I tried:

SETF Create keyframe for all vbls at their current value

CHGK Chg frame number of a frame/value pair

INSK Insert frames. Adds constant to all keyframe nbrs after current one

SCLK Stretch scene. Multiply keyframe nbrs by scale factor.

DELK Delete keyframe/value pair

ADJF Connect all vbls having current keyframe entry to knobs, allow adjustment of knobs and update keyframe table.

Ways that worked best

FET display Time plot of keyframe table

Screen editor Like token list editor but for keyframe/value pairs.

These last two will be more fully described in the next two sections

Viewing

Several ways to view keyframe table, sometimes called a Frame-Event-Table, or FET.

1. playing back animation
2. printing table sorted by variable name
3. printing table sorted by keyframe
4. FET printout: by constant keyframe increment, by keyframe values
5. FET display on PS, like a piano roll

This last was, by far, the most useful. It looks something like

```
B     |=====|
A        |======|
```

It shows a horizontal box for the frame number range over which a symbol changes. A vertical line shows current frame, parallel lines on left/right are markers for ± 1 second from current frame. Can change x scale, y position of this display.

Both wireframe of the current scene and the piano roll are viewable on PS. They are selected by a switch setting, can put both on at once. This sounds confusing but it's real useful, It doesn't take much time to train the eye to 'see' only that part you're interested in.

General philosophical point. It is useful to have various ways to view a database. This also applies to ordinary computer programming. Normaly you only look at a listing of code, but it's useful to use the global scan/print command of a text editor to see all usages of a particular variable. Can compare usages and sometimes see bugs you wouldn't have otherwise.

Editing Keyframe Table

The most effective keyframe editing technique proved to be a mode similar to the token list editing mode—the spread sheet metaphor. In this mode ther is 1 column per symbol, 1 row per keyframe. The user enters this mode via a EDIF command and can list up to 6 symbol names he is interested in.

The command forms the union of all keyframe values defined for the 6 variables. This forms the row labels.

When displaying the table, any cells that correspond to nonexistant symbol/keyframe values are left blank.

Example `EDEF A,B,C`

The screen looks like

```
          A     B     C
150             2
200       4
275             3
300     8.2
325             5
600       0
```

Now the user can move the cursor from cel to cel and modify the table. If the cursor is moved up/down a new wireframe display is

created corresponding to the frame number for the row the cursor is on.

When the cursor is in a cel, typing a number will replace the value there and update the keyframe database accordingly. If the cel was originally blank (no keyframe there) typing a number creates a new keyframe for whatever symbol the cursor is in the column of.

Moving the cursor to the keyframe label cell on the left column and typing a new number will change the keyframe field of any symbols which have a keyframe on that line. This is useful for tuning the timing of a set of keyframes.

The user can then enter an assortment of single key control keys to perform some other functions that were found useful. These were selected to have similar functions to the control key commands in our text editor, since I have esablished the rhythm from using it. These functions were determined by need, according to the principle of really-needing-it before implementing it, and the principle that what you do most often should require fewest keystrokes.

Single key commands include

- Increment/Decrement the value in the current cell by a delta amount
- Increment/Decrement the delta amount used in above command These commands are typically used to position the objects in the first place. Placing them via tablet just doesnt seem natural to me.
- Insert a new frame number in key list (column labels on left side) This is done either 20 frames past the keyframe of the current cursor location, or halfway between framenumbers of the cursor locatin and the one below it. All cells in this row will quite likely start out being empty but can be filled in later.
- Change the column labels, changing the symbol displayed in that column
- Evaluate the value of the symbol at the current cel. This is meaningful only for cels that are initially empty. It creates another keyframe having the value that the normal interpolation would have been. It can then be adjusted up and down.
- Delete a keyframe/value `ctl-D`

This technique has proved to be very valuable in fiddling with large and complex lists of keyframes. It also has had some unexpected

uses beyond editing. For example, it is sometimes useful to step through an animation flip-book style. Setting the desired keyframes to flip between, and moving the cursor up and down does this. It is also useful to be able to compare two keyframes at widely different locations in the animation. You can define a dummy, unused, variable having keyframes at just the two desired frames. Then go into EDIF mode and flip by moving the cursor up/down.

Interpolation techniques

Interpolation is performed by putting a cubic polynomial between each key/value pair. The keyframe values are the endpoints, the end slopes are generated by various techniques. A flag in the table for each symbol tells the technique used throughout, for that symbol. Techniques are

Linear

INOUT (zero derivative at each keyframe)

SMOO (Catmull-Rom interpolating C1 continuous spline)

It is also possible to explicitly specify a slope at a particular keyframe which overrides the automatically generated one. This is done by the special syntax of putting a prime after name e.g.

```
EASE INOT,A                "sets flag for A to INOUT"
KEYF A, 1000,0, 1100, 1, 1200, 0
KEYF A', 1100,.1
```

The slope of A is zero at frames 1000 and 1200 according to the global flag, and is .1 at frame 1100 according to the explicit specification.

You can also pick A' as a column in EDIF mode yeidling a spreadsheet that looks like

	A	A'
1000	0	
1100	1	.1
1200	0	

There is also a program that plots a symbol-value vs. frame-number graph. It includes the same keyframe table editing module for reading,

editing and saving a .MOV file. The display on the PS is just a graph of a symbol instead of a wireframe look at the scene. It is actually quite interesing and educational to type numbers for the value and slope of a variable in EDIF mode and see the graph of the resulting interpolated curve.

Playing the movie

Note the difference between PLAY, which copies new values into the symbol table, and ADJ, which fiddles the symbol table into a state that doesn't necessarily correspond to any particular keyframe.

Re-ececuting a PLAY command snaps the picture back to the keyframe for those symbols that are in the keyframe table. Sometimes it's good to put in constant keyframes for a variable to guarantee that it won't be changed accidentally.

The FREZ command temporarily disables update of a particular symbol. It's used for fiddling.

7.6.11 Table look up of parameters

Some simple generalizations of the above mechanism can provide a wealth of new capabilities. I don't want to just throw them in, however, so before I describe what they are, let's look at some real problems.

Sometimes there is some simulation that can be explicitly written as a function of time, but you need to play it back in fits and starts. To freeze the simulated time, do some algebra, resume the time sequence, freeze it, display some label, etc. Computing the starting and stopping points is a nuisance.

Sometimes there is some sequence of events that you want to repeat, like a walk cycle. Replicating the table for each iteration of the cycle is a nuisance.

These can be done by realizing that the movie keyframe table is just a function definition. Animated variables are essentially function calls where the parameter—frame number—is implicit.

These have been made explicit by the following simple additions.

First, the keyframe table is made to allow floating point frame numbers. A table for a given symbol can define a function over any floating point range of values.

Second, a version of the simple S*a+b evaluator for parameters allows an alternate syntax: S(T)*a+b. When this form is evaluated we take the current symbolic value of T, use it as a parameter to the movie table for symbol S, finally scale and offset it by a and b. Then the starting and stopping happens by animating the variable T.

This is usually designed by starting with

```
PUSH
TRAN  X,Y,0
DRAW SOMETHING
POP
```

Design the animation for one cycle of motion via editing keyframes of X and Y. Then edit the .SKLfile to be

```
PUSH
TRAN X(TIME), Y(TIME), 0
DRAW SOMETHING
POP
```

Then make the animation start/stop by editing keyframes for TIME. A mechanism described below enables you to make a cycle by calculating just the fractional part of TIME and using it.

7.6.12 Algebraic Expression Evaluator

After a while, there came to be too many instances of need for expressions more complex than S*a+b. I finally caved in and wrote a general expression evaluator, but its use is somewhat unusual. Firstly, since Fortran can't do recursion, an expression compiler is too difficult to write. Expressions are therefore entered in postfix notation (designers of languages like PostScript or Forth have been getting away with this for years). This makes them easy to tokenize and detokenize. Secondly, they still cannot appear anywhere in numeric parameters. They operate essentially as a list of assignment statements, which evaluate expressions containing values from the symbol table, and store the results in some other symbol in the table. This list is executed in one burst just before calling the .SKLinterpreter, on each movie frame execution. The reasons for doing it this way are several. One is just the implementation convenience. Allowing arbitrarily long expressions in parameters still would mess up the screen

editor mode. Many times the result of an expression will be used in several places, calculating it anew upon each occurrence seems wrong. This means some sort of assignment command would have to be added to the .SKLinterpreter. Some form of assignment statement would be useful anyway. I really didn't want to make this look so much like a programming language (but I seem to be losing the battle). I wanted to make a general purpose evaluation module that could be perhaps used other places (an 'object' if you will.).

So the internal operation of the expression evaluator module operates as follows. It has entry points to

- Interpret keyboard command input (as described under CLI) Keyboard commands enter expressions in the expression list.
- Evaluate the list of currently stored expressions
- Write (save) the current list to some output channel (for the .SKLsave operation)

It calls only the symbol table routines:

- Lookup symbol in table
- Get current value of symbol
- Set current value of symbol

This module could then easily be incorporated into other programs.

Arithmetic consists of the obvious arithmetic functions $(+ - */)$ and a few usual trig and exp type functions.

In addition there is another function built in, an evaluation of the funciton to convert mean anomaly to true anomaly as described in an earlier chapter. This function is used in plotting keplerian orbits as functions of time.

Also a fractional-part function. This is very useful for cycles. Example

```
EXPR CFRAC, CYCLE;f  "eval frac part of CYCLE"
                     "store it in CFRAC"
....
TRAN X(CFRAC), Y(CFRAC), 0
....
```

One cycle is defined by the keyframes of X and Y , which are floating point numbers between 0 and 1. The cycle is run by animating variable CYCLE from, say, 0 to 25. The fractional part will be used to look up the postions for the object.

Also included is the ability to define and evaluate arbitrary functions via table lookup. This uses the movie table routines to define the function and the movie table interpolations to evaluate them.

7.6.13 The right way to do the above.

The last two sections imply a bit of nonuniformity and kluginess in the evaluation of expressions. This is probably true. Perhaps the best way to do this is to allow arbitrary expressions and add an assignment command to the token interpreter. This would allow parameterized subassemblies since all symbol names are global. (note: SET is already used for an entirely different purpose, so postulate an assignment token called LET)

```
LET  A,B1+2./SQRT(F)
DRAW PARTHING
LET  A,C+D+ECP(QRST)
DRAW PARTHING
 . . . .
DEF PARTHING
PUSH
SCAL  1,A,1
DRAW SQUARE
POP
----
```

(oh no we've reinvented BASIC). I might do this sooner or later, but I'm not sure about any subtle side effects. I'm still worried about how to cram a complex expression into a screen editor cel. I know the current scheme works, I just haven't tried this postulated more uniform one. I'm sure that most filter-style animation systems do it this nicer way, but part of the trick is to keep the interpreter quick and the editing of the 'program' lively.

7.6.14 Cleanup commands

We aren't just feeding perfect, canned routines to ARTIC. We are constantly fiddling with them, constantly adding/deleting stuff. Here

are some commands to garbage collect old, obsolete ideas.

ORPH finds entries in symbol table that nobody references and flags them. Can automatically delete them. These can show up due to typeos or deleted ideas.

CLEA deletes redundant keyframe table entrys, i.e.,

```
Key Val
100  1
200  1     <- deletes this one
300  1
400  2
```

Appropriately different criteria are used if the SMOO interoolation is in effect for a particular variable.

7.6.15 Special purpose versions

Several types of simulation required special purpose extensions to the animation program. Here we discribe the hooks left to do this. Special purpose versions may be justified for the following general reasons

- There are lots of closed form calculations to do. This could be done with lots of EXPR evaluations but it's clumsy.
- An object changes shape. The shape may be parameterized by only a few numbers. For each frame of an animation, the simulation program needs to write out some simple rendering database, as well as the .TMP files.
- Some numerical simulation of the 'particle pushing' variety.

This breaks down into two general types

1. Keyframe interpolation. Each frame is determined solely by interpolation between keyframes. Can jump to any frame as easily as any other.
2. Incremental. Simulation needs state-information as of frame N-1 to calculate frame N.

These special versions are implemented as a subroutine package that is linked into `ARTIC`. Calls to routines to perform initialization, parameter entry, and parameter query are part of the `ARTIC` loop.

First let's look at a simple example.

Reveal

The simplest special purpose hook is for an animation technique I will call here the 'reveal'. This is, in fact, a default special version that is built into the ordinary `ARTIC` program. Other special versions consist of replacing this module with some other code having the same entry points but that do different things.

The reveal operaion means 'draw only the first R segments of a primitve'. This is useful to animate things being drawn out for you on the screen. It is so common that REVL is a command present in many FB rendering programs. There is also a `.SKL`token to set the reveal count.

The REVEAL module contains entry points

INIT

NEWFRM call

DRAWPS called to draw primitive on PS

DRAWFB called to write command to .TMP file that will cause primitive to be rendered

These routines can call any of the following sets of routines to implement their will

- Symbol table creation and enquiry routines
- PS low level drawing routines
- can write to the currently open .TMP file

Procedural primitives

How is this interface used for other special purpose programs?

It has access to the symbol table. It can calculate values for some or many symbols and store them in the symbol table. It can get its input parameters from the symbol table, which may be animated. It might do only this, i.e. it might have no special primitive.

It may define some primitives of a third type, procedural. (We already have subassemblies and primitives, now we have those two and a procedural primitive). A given entry point is called whenever the `.SKL`interpreter desires to draw such a primitive. This entry point then does the procedural modelling and calls whatever drawing routines are necessary to draw the result.

Particle pushers

An example of a particle pusher is the Lennard-Jones simulation. This type of simulation is characterized by a largish database to represent state (around 100 values). It requires the state at time F to be able to calculate the state at time F+1. A simulation step counter keeps track of how many simulation steps have been performed. The step counter is then part of the state information.

Problem: To allow external events (e.g. view direction, positions of walls) to be animated at certain steps of the simulation. To keep track of step number and allow restarting.

General operation— read file of initial conditions and set step nbr=0. Keep WANTSTEP as symbol in keyframe table. This is animated to advance as desired as a function of frame number. Whenever the draw-frame routine is called; check WANTSTEP against internal step number. If internal step is less, advance the simulation to catch up. If internal step has already passed WANTSTEP, print error. (You could imagine simulating with negative time steps, but its a bad idea. Numerical inaccuracies will screw you.)

Keyboard ommands to generate, edit and save initial conditions. Simple command to run simulation forward for testing purposes.

Examples

- Lennard-Jones simulation and variants. Brown/blue atoms. Separately displayed momenta. Hole in box for Joule-Thompson effect. Electrons and crystal imperfections version.
- Ideal gas law
- Could have one for gravity but kludged it. Used a totally different program that was menu interactive, kludged to dump calculated positions on a file in KEYF command format. Also used similar version for original LJ animation
- SWIRL randomly placed initial positions of marker particles. Analytically calculated time evolution under velocity field.
- EMWAVE Analytic solution to shape of electric field lines as charge oscillates with sine wave.
- WATER
- WAVE

Funny story. Subtle dependance on initial conditions. Originally saved state as 5 significant digits. On one scene I designed initial conditions, saved them and started making frames without exiting program. Later tried to restart by reading initial-conditions file. Came up with different frames, because 6 and 7th significant digits in initial conditions have a remarkable effect on the state only a few hundred time steps into the future. Had to do scene all over again. Maybe that wasn't so funny after all.

7.6.16 Wrap-up

We have the following databases

- A token list broken into subassemblies [SKEL]
- A list of PS commands for primitives, broken into separate primitives.
- A dictionary of names for the above two.
- A symbol table of floating point values. [SYMTAB]
- A table of Levels and textual information on how to generate rendering commands for them. [LEVL]
- A list of keyframe/value pairs for any animated symbols. [MOVIE]
- A list of algebraic expressions to evaluate at each frame. [EXPR]
- A list of knob/symbol connections for the ADJ command. [KNOB]

Ways to edit:

SKEL type in new stuf with DEF or ADD2, screen edit mode

SYMTAB Keyboard cmds to SET,

MOVIE keyboard commands KEYF, screen editor, KREC records knob settings, various cmds to record current state of symtab into movie tbl, RECO from tablet (tried but havent used much, Fills up table real fast, Need ways to filter out redundant info)

EXPR keyboard commands only add to list

LEVL declaration table keyboard commands to update fixed slots in table

KNOB table keyboard commands update table for each knob pointing at a symbol

Operation of SKL interpreters

To DRAW on PS

1. call movie table interpolation routine
2. call expression table evaluation routine
3. call skeleton interpreter
   (a) call SYMTAB routines to get current values of any required symbols to evaluate parameters
   (b) translates tokens to PS commands
   (c) buffers them out to PS

To DRAW on FB

1. call movie table interpolation routine
2. call expression table evaluation routine
3. for each level declared in level table
   (a) open Ln.TMP
   (b) set LWANT to n
   (c) call skeleton interpreter which
      i. calls SYMTAB routines to get current values of any required symbols to evaluate parameters
      ii. translates tokens to text strings and writes to .TMP file
      iii. if DRAW,...., or SET command: only writes if LEVL matches LWANT
4. spawn task to execute X.COM

Types of files

.SKL Token subassembly lists

.MOV Keyframe table data

.ART Main file containing global setup commands and READ commands for above two files.

.COM VMS commands to execute desired rendering programs when invoked by DRAW command.

These 4 files are necessary for each animated scene. Initial versions can be created by a canned command procedure. Text editor can be used to tailor contents for special situations.

Since many scenes are modifications of other scenes, there is a canned procedure to copy all four to different names, and edit any internal old name references to the proper new name references.

Overall

All techniques not new or radical. The trick nowdays is to see how to fit them together, how to make them talk to each other.

Chapter 8

Scene Implementation

Now that we know what we want, and have the software to do it, let's get to work. But first, a few words about some global organizing strategy.

8.1 Data location

There is a lot of data involved in 550 scenes. This section discusses some conventions on storing that data so it doesn't get lost.

8.1.1 Physical Data

Production of animation requires actual sheets of paper in addition to databases on a computer. Keeping track of these sheets of paper doesn't seem like a major techlological issue at first, but there's a lot of sheets in a project of this size. The blizzard was tamed by the simple convention of allocating a 1" ring binder for each program, carefully labelled on the back with the program's name and number, all placed on a shelf in one big row. In the binder was kept

1. The current version of the script for that program.
2. Storyboard sketches for each animation.
3. Mathematical derivations for any Physical simulations needed in that program.
4. Copies of pages of books referenced to develop the simulations.

Why am I making such a big deal about this? Because it was so useful. I was amazed at the chaos in keeping track of my notes early in the project. This simple organizing principle improved my life considerably. You can't have "a place for everything, and everything in its place" if you haven't allocated "a place" for some things.

This is one example of a major trade-off that appears in the conclusions section of these notes: anarchy vs. beaurocracy. Forcing yourself to be a bit compulsive in some large scale organizational ideas really helps in the long run.

Also, as each program was completed, I glued a gold star on the back of the binder for that program. Over the four years of the project, it was gratifying to watch the shelf of binders slowly accumulate gold stars.

8.1.2 File naming conventions

It is usually a good idea to give files descriptive names. This wound up being a bad idea for the scenes of MU because there are just too many of them. Remembering a mnemonic name is harder than remembering a numeric label for a scene number.

We must make up names for the following types of files.

- `ARTIC` command files: x.ART, x.SKL, x.MOV, x.COM
- Any .TMP files written by `ARTIC` to communicate with a rendering program.
- The picture files for each frame of each animation

As far as the software is concerned these could all have different names, but it is useful to have some organized naming convention. Nothing in the animation software either prevents or encourages these naming conventions. It's just a personal design convention.

Prefix Generation

It seemed like a good idea to have a unique name for every picture the entire project. Picture files created by the animation program have names that begin with some letter sequence (called a *prefix*) followed by the frame number, like `JB1045`. The following naming convention was used to generate prefixes for each scenes.

1st letter	A...Z	programs 1...26
	AX...ZX	programs 127...152
next letter	A,B,C...	scene numbers 1,2,3...
next letter	A,B,C...	subscene letters A,B,C

A picture file named JAB1050 could be easily recognized as coming from program 10, scene 1B, frame 1050.

The control files, like .SKL files, were given the name JAB.SKL, JAB.MOV etc.

The old way

Actually for MU1 this was not done so cleanly. For example data for scene 1B would be files:

SC1B.ART ARTIC commands and pointers to MV and SK files

MV1B.ART Movie table generating commands

SK1B.ART Skeleton transform commmands

SC1B.COM VMS commands to spawn to render a frame

This was because the first three files are essentially all commands to ARTIC. This convention proved unpleasant because picture files and the database to generate them had different names. Also, scene 1 of program n had the same database file names as scene 1 of program m.

Evolution

Why subscenes? Usually, in breaking down a program, the first pass requires Scene1...SceneN (where N=4...7). But, in adjusting the design, some are dropped and other complex ones split into several pieces. So have subscene numbers.

Temporary files

The temporary files written by ARTIC to communicate with rendering programs have names constructed with the scene prefix followed by L1 for LEVL 1, L2 for LEVL 2, etc. This means that you can't run production of different frame number ranges of the same scene on two different terminals. This is because they would both be using

the same temporary file names. This has not been a major problem. On some cases where it is really necessary to run simultaneously, I just made a temporary copy of all 4 files with some new name. The picture files can be made to go to the correct file name because the MAKE command takes any prefix string you type in.

The system could be instructed to make up unique names. There are subtle inconveniences attached to this. The temp files stay around after a frame is made, so you can look at them and re-execute them, if you know their name. While designing and debugging a scene you can re-make a frame just by saying, for example, `@JAB`. You can re-make just certain levels by typing whatever command does that level in JAB.COM, such as `$FBDRAWS READ JABL3.TMP`.

More temporary files

Some programs have to write out other temporary files, like sorted polygon lists from PY123 or like files describing the shape of some shape changing object for a special case version of `ARTIC`. These files are created with names unique to the process that starts them. This is because the same program might be running on two different terminals doing two different scenes. While it not too much of a nuisance not being able to do two different fram number ranges of a *single* scene simultaneously, it *is* a nuicsance not being able to simultaneously do two *different* scenes that use the same *program*.

8.1.3 Directory Allocation

A directory was created for each program in the series (that's TV program, not computer program). For example, `[BLINN.MOVIE.MU.E2]` has all data for for program 2. The first half of series has directories named `E1...E26`. The numbering was changed in the second half so it's `E127...E152`. Most all data for animations for a particular program is in the corresponding directory. The first and last programs for each half are Intro and Wrapup and so have no new animation.

8.1.4 Shape Libraries

There are several library directories. These are places where a rendering program or `ARTIC` itself looks for data if it is not on the current directory. The largest of these is a shape library called `[BLINN.MODEL.LIB]`.

This contains things like .LIN and .PS files for many common shapes. Files specific to the MU project had `MU` as the first two letters. As shapes were needed that were used in more than one program they were moved to this directory. It got real big. It currently has a primitive for almost every letter and digit.

There is a design trade-off here; distributed libraries vs. all data in one place. The advantage to distributed libraries is *information hiding*, the disadvantage is that you don't know where everything is when moving to another machine.

8.2 Transformation Tree Design

The first thing to do in actually implementing a scene is to generate the .SKL file. This is sort of like programming, but on a smaller scale. You would like to be able to look ahead and group things into subassemblies according to how they are going to move. There are two problems. One, the concept for the motion of the scene might change as you are designing it. Two, the visual groupings sometimes need to change during the motion of a scene.

8.2.1 Algebraic Ballets

As a good example of transformation design, consider the algebraic ballets.

I started out using a simple typesetting program to make whole expressions as primitives. But wound up needing to move subexpressions independantly. So I made each letter a primitive and grouped algebraic terms into subassemblies. But then found out I needed to move the components of a subexpression separately. So now I make each letter an assembly. If two must move together for a while, their individual x, y coordinates are animated. Since transformations are easy in the `ARTIC` system, scenes become very transformation intensive.

example

(x+y) moves together then later in the scene each moves separately.

8.2.2 Post facto regrouping and splitting

If the initial subdivision of components of a scene turns out to be inconvenient, it may become necessary to group items or split groups after a scene is designed. This can be done by saving the .SKL file and modifying it with the text editor. (This is another argument for using ASCII databases instead of binary databases). This may sound like a nuisance, but it is a very powerful tool. Many iterated operations that text editors are very good at would be enormously complex to implement within ARTIC with the database in place.

8.2.3 Variable naming

Although you can use any variable names you like, I find it convenient to have the first letter of a name indicate its usage if possible. First letters would be

X X position

Y Y position

Z Z position

A an angle

O an opacity

L a level

In general, with a complex scene there might be several hundred variables all being animated. This can lead to anarchy. Its necessary to keep a labelled sketch telling what variable controls what object. It might seem that some automatic 'hit detection' might be necessary, but I find it pretty quick to just glance down at a piece of paper with notes if I forget the name of a variable controlling something.

8.3 Timing design

8.3.1 Starting point

Scenes are designed starting with frame number 1000. There are two reasons for this. First, a directory listing of the picture files will be sorted in the correct order. If we had started with frame number 0 the frames would be named X0, X1, X2,... X10, X11,... X100,X101,...

When sorted by file name, they will appear in the order X0, X1, X10, X11, X100, X2 ... Secondly, if an after-the-fact modification is made to the scene, requiring new frames to be inserted at the beginning, it is still possible to call them X900 X901 etc. This sort of thing turns up being necessary more often than you would think and it is very valuable to allow for it.

8.3.2 Spacing

Scenes are designed by laying out gross motions in steps of 20 frames. This is just to get the sequence of events down for evaluation. Then I go back inserting and deleting space between keyframes to adjust timing.

8.4 Image rendering tricks

Example of book page and text as example of how Z buffer wouldnt help.

Anyone who attempts to simulate complex physical phenomena soon realizes that literal simulation is far beyond the capabilities of today's hardware or software. If fact, such literal simulation is not really necessary to make adequate pictures. All that is necessary is that the errors be confined to situations which are not visually apparent. This may involve generating models that only work in the geometric situations of interest, but break down when view from an un-planned-for direction. It may involve mechanisms where manual intervention is required to perform some global decision making about visibility. It may involve some numerical approximation that generates the wrong picture, but it is not wrong enough to be immediately apparent except with detailed examination of the image.

Many of these at first seem like very special purpose tricks, but, as we all know, a "technique" is a "trick" that you use more than once.

8.4.1 Modeling

Variable Resolution Models

Changing the resolution of the database based on number of pixels covered is a fairly standard trick. This is known as "the old switcheroo".

Many of the moons of a planet are far enough away to only cover a fraction of a pixel. The main control program of the space scene renderer either draws a texture mapped sphere or a simple anti-aliased dot. The pixel size at which this transition occurs is set manually after making a few test frames. The color of the dot is also manually chosen to be approximately the same intensity of a sphere at the same size.

A similar database/rendering switch was used in a scene depicting a flight over the moon Mimas. When situated near the north pole it was rendered with a bump mapped sphere. When near the large crater at the equator it was rendered as a brute force polygon mesh. The transition was made manually by first generating the frames near the pole, stopping frame production, editing the control file to cause it to invoke a different rendering program, and then re-starting production. The transition was done when flying over the night side of the moon where the image was too dark to notice the slight change in appearance.

Other examples of switcheroo are

- An exploding 8-ball was used to show center of mass motion. Before the explosion it was a sphere. Afterwards it was modelled as a collection of polygons.

Texture Pattern Generation

Texture maps are stored and utilized as pixel arrays. This means the rendering program is insensitive to the kluges used to generate these arrays in the first place. All it takes is for the rendered image to look right.

Many of the moon maps were re-constructed from real photographs of the moons sent back by an earlier spacecraft. This basically involves constructing an effective viewing transformation for the real image, transforming pixels in map space into the pixel space of the real image (via this transformation), picking up the intensity of the real image, and finally placing it in the texture map. (sort of like running a rendering program in reverse). The viewing geometry is known fairly precisely at the time the image was made from spacecraft orbit data, but some adjustment must always be done. The image is never seems to be centered correctly, so an interactive program was used to fine tune the reconstructed viewing transform so that the simulated silhouette edge of the moon matched that on the real image. Further, when this 'unwrapping' process is completed, only about half

the map is generated (the half facing the viewer). The map still has the effect of the light source direction scaling the intensities however. Moons are not perfect lambert reflectors so some empirical juggling was done to construct effective lighting functions which, when divided into the picture, fairly nicely flatten out the global gradations in shading across the map. Next, the data around the edges of this covered area is usually pretty ratty since it comes from a stretched out part of the image. Several overlapping real pictures are needed to completely cover the moon. These never seem to have matching intensities at their edges. This is manually fiddled with to get them all about the same brightness and then they are blended together with weighting function adjusted to disguise the seam between pictures. Finally, some of the moons were not completely covered with pictures. The simple expedient of "cosmetically enhancing" the map was done by using a painting program.

For texture maps that are completely hypothetical, some other lighting tricks were employed. Much of the lighting of a moon is caused by surface irregularities. This might be ideally modelled by bump mapping. The problem is that bump mapping is a bit slow. Also the craters or mountains tend to be rather small creating bump aliasing problems. Finally, the artists employed were more used to thinking about painting illuminated surfaces rather than sculpting topographic maps. For these reasons, the artists typically painted the shadows of craters etc. right into the maps. The moon was then rotated so the sub-solar point (as painted) pointed in the direction of the sun (as modelled by the orbit calculations). It looks fine for that particular scene, but a bit fishy if a simulation is done for a different time.

Intensity Modelling

THe concept of realism has to be stretched slightly when making space scenes or they will look rather boring. For the most part, the objects in a typical scene have radically different intensities. To simulate Jupiter with its intensity scaled to 0 to 255 intensity values, the moons would be almost dead black. Likewise, the spacecraft are typically enclosed in light absorbing cloth to improve their heat retention cababilities. Therefore the relative intensities of most objects in the scene are adjustd independantly to make them all show up at a similar intensity.

Another thing to remember is that there are no fill lights in space. The whole scene is "really" illuminated by one bright point light source.

This is ideal for computer graphics but looks too harsh to be able to percieve shapes. This is especially a nuisance when you realize that half the viewing direction you can pick are on the night (black) side of an object. Thus, for viewing ease, some ambient and fill lights were usually added to the simulations.

Specialized Primitives

Most simple objects can be modelled with polygons, surfaces of revolution or extrusions. The Voyager spacecraft, however, is built up of a large number of thin struts and booms. A special modelling primitive was built into the polygon rendering program to model tubes directly as beginning-point, ending-point, and radius. This is converted into a standard unit cylinder with the appropriate orientation and scaling parameters. Then during rendering, each cylinder is converted into two long thin polygons. Their outer edges were calculated from the mathematical silhouette outline of the cylender (in screen space). The common edge between them is calculated as that line where there is a local maximum in intensity when scanning across the tube. Thus the cylender was modelled with two optimally chosen polygons. This is actually a fairly simple 2D calculation.

This works fine and saves polygons (important in a memory restricted environment) but breaks down when you get too close to the tube and notice the strange appearance of their ends.

Modelling with Transformations

The main animation system currently in use allows for a collection of primitive objects to be subjected to an arbitrary set of nested homogeneous transformations. It is not good at shape alterations. One can, however, do more with matrices than might be expected.

Viabrating String For example, a viabrating string was animated by beginning with a (rigid) primitive flexed string, and scaling it by zero along the perpendicular axis to make a straight string. Then animating the scale factor back and forth between positive and negative values caused it to twang.

Bending Vector Similarly, a vector arrow was made to appear to bend by modelling it bent and scaling it by zero when it needed to

be straight. The arrowhead was modelled as a separate object and animated to follow the body of the arrow as it flexed.

Conic sections Another particularly interesting example is the illustration of conic sections with variable eccentricities. These can be generated by using the homogeneous perspective transformation to distort a circle into any desired conic. Given the polar equation for a general conic:

$$r = p/(1 + e\cos\theta)$$

where e=eccentricity

one can generate the conic by transforming a circle formed from sines and cosines by the perspective matrix:

$$[\cos\theta, \sin\theta, 0, 1] \begin{bmatrix} p & 0 & 0 & e \\ 0 & p & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A perspective transform is therefore used in two places, once in the normal sense to view the entire scene, and once, as above, as a modelling transform.

8.4.2 Rendering

Geometric Calculations

Problems with Temporal Priority

1. Z sorted line renderer
2. Saturn ring splitting and making sure halves dont overlap
3. Jupiter mag field
4. Transparent spinning top with internal structure
5. Spinning gyroscope wheel with supered force vectors

Intensity Calculations

Shadows

- Explicit drop shadows of letters,
  - 'local' color and 'global' color
  - shadows of solid objects on flat surfaces
  - perspective (local light) shadows
- Rings of saturn
- Planet on rings
- Satellite shadows (black dots)
- Eclipse shadows, approx with spheres, umbra/penumbra

Depth perception enhancement

- Fog simulation (avoids drawing too many spheres of NaCl)
- Darkened edge lines (jup mag field)
- Playing with depth cueing (jup field)
- Shading background (vectorland)
- 3 levels of glow on zooming into nucleus
- Explicit fade outs as objects receed

Textures Bump mapping problems at terminator

Soft objects

- Puff of smoke (scaling up and fading)
- Explosion (scaling up and fading)
- sparkles (fuse on 8 ball)
- fuzzy lines overlapping
- earth atmosphere on MU open

8.4.3 Anti-Aliasing

Geometric

- Subsampling
- Edge blending
- Ring shadows
- Sleazy lines, limitations with wide lines

Textures

- Subsampling
- Global pre-filtering (removing hi frequencies from pattern)
- Specialized pre-filtering based on shape the texture will be mapped to (e.g. spheres)

Temporal

- Streaks of ring particles (comet shaped, rather than Gaussian)
- Time smearing texture patterns
- Trails of moving particles as explicitly drawn lines
- Logarithmic zooms
- When to double/single frame

Chapter 9

Video

Film sux. It is much easier to record directly to video since the final result is going to be video anyway. We were lucky in that the technology to do this reliably became available just as the project was starting.

9.1 Generating a Video Signal

This section will just give a sense of what needs to be done to make a broadcast quality video signal. Sometimes the best thing to know about a problem is simply how hard it is. It seems that video is easy to generate, but it's not. The machinations experienced here will show what you might have to do in another system

9.1.1 Properties of the NTSC video signal

- Timing based on 3.579545Mhz ±10Hz color carrier
- 525 lines, 485 visible
- 29.995 frames/second, 2 fields/frame
- SCH phase
- color burst

9.1.2 RGB to NTSC Conversion

- multiply RGB times matrix to get YIQ
- Y is luminance
- I and Q modulate color carrier, sin and cosine versions.

9.1.3 Synchronizing Computer Display to NTSC

Computer displays are looser. Monitors dont care so much about timing, video recorders and broadcasters do. Computers displays have their own line rate that's probably not the NTSC standard. You first need to adjust crystals to get computer display close to correct timing.

Then you need the computer display to respond to an external synch signal. This is a process called Genlock.

An external sync-generator generates a highly accurate basic signal. Outputs of a sync generator are:

1. Subcarrier, 3.58...MHz color carrier.
2. Horizontal Drive, 4 volt pulse at each horizontal line end.
3. Vertical Drive, 4 volt pulse at each field end.
4. Blanking, 4 volt signal enclosing H Drive used to disable video portion of signal.
5. A composite signal, a black screen with all timing signals included. This is sometimes called "color black" (a curious phrase).

Our display, a DeAnza, can genlock to either a color-black signal or to the HDriv, VDriv pair.

Frequency response

A computer display generates square edged pixel transitions. These contain high frequencies that disturb RGB encoders. It is necessary to connect the DeAnza to the encoder through bandpass filters with cut-offs about 4.5 MHz. The exact frequency was chosen more for the availablity of the parts than for theoretical considerations.

Encoder

Had to remove sync,blanking on RGB so not to confuse encoder. This required modified traces on pc board in the DeAnza.

Time Delays

Time delays due to genlock cause RGB in to encoder to be delayed, so pictrue is shifted horizontally. Extra time delay due to bandpass filters on DeAnza output. These were removed by placing 'pulse delay lines' (tuned for 4 volt square waves) between Sync Generator and Encoder on the following lines: HDriv, VDriv, Blanking. We needed about 3300ns delays here. This is about ? cycles of the subcarrier. So we didn't bother to delay carrier. There is a screwdriver adjustment in the encoder that is used to get phase correct. The BVH 2500 videotape machine has a 'SCH meter' to use to tell when phase adjustment in encoder is correct.

Zone plate good test pattern. This is a pattern with line n containing n cycles of a sine wave. Line 227.5 is the frequency of the color carrier.

Hum Bars

There are problems with 60 hz noise getting into the video. Since the refresh rate is 29.995 Hz, there will be a beat of several seconds. This looks like a dark bar crawling slowly up the screen. It might be very subtle and hard to see normally, but if you animate with such a system, you are recording frames about every 5 seconds. When you play back the tape, the bars move fast enough to be obvious. To avoid Hum you need good grounding, isloation transformers, separate power for noisy equipment, a rabbits foot. It's amazing how dirty power can be and most equipment is unaffected. (3 phase balancing)

9.1.4 Connecting signal

Now have good signal, what to do with it.

Can't treat like ordinary electricity; don't use T connectors, these cause reflections in the signal. Use 75 ohm (not 58 ohm) cables. Terminate all signals with 75 ohm terminators. Most good video equipment has pass thru connectors to use to 'daisy chain' a signal through several devices.

9.2 Designing for Video

Most TV screens do not show the entire video picture. They can cut off up to 25 percent of it at the edges. It is necessary to design scenes

so that no vital information is within 1/4 of screen width of the edges. This is called the 'TV safe' area. An overlay is built into `ARTIC` to show this area for design purposes.

9.3 SMPTE Timecode

- Longitudinal. Used most commonly, it works ok.
- Vertical
- Drop frame

Time code readers are pretty smart these days, so can keep in synch.

9.4 Operation of 1" videotape machine

- Helical scan (hi freqs need fast head-tape speed) with spinning heads. Timeouts to stop heads if no tape motion after certain time. Horror stories about what happens if heads spin too long on one spot of tape.
- Longitudinal control track. Pulse every field, to use for alignment of helical scan during playback.
- 3 audio tracks. One used for Longitudinal Timecode.
- Synch track and heads. Historically not needed, resynthesized by TBC. Nowdays scan lines during vertical retrace have info,so need to record.

Video is a set of evolving standards, lots of options on tape equipment to allow for reading old stuff.

2500 needs reference synch. for playback synchronizing with other signals, we just jumper thru from video in.

9.4.1 time base corrector

Slight fluctuations in speed cause big phase distortions in color carrier. TBC is digital device to correct this. Digitizes signal into memory, another process analogizes it out at an automatically adjusted rate to correct.

9.4.2 Dynamic Tracking

Piezoielectric crystal moves heads sideways to find the helical track, even if tape is not moving at standard speed. Allows slo motion playback without distortion of picture.

9.4.3 Recording

Three types of recording

Assemble

Writes new control track, used to initialize tape.

Insert

Starts out in playback, synchronizes scan properly. Then kicks over into record for some number of frames. Only records picture data, leaves control track intact. Most stable method of changing a frame in the middle of something.

Δt

Sony's brand name for recording single frame without moving tape. Because tape isn't moving head to tape speed is not correct. Uses digital time base 'decorrector' to record so signal is correct when tape plays back in motion. Works pretty well but blows it now and then. Have to find bad frames and re-record.

9.4.4 Preparing tape

Must record control track over whole tape. Record blank screen with time code generator running, then NEVER record with 'assemble' again. Only 'insert' or 'Δt'.

9.4.5 Computer Connection

IEEE RS422 connector in back talks to any computer. Has faster clock speed than RS232 connector. Commands are just binary bytes. Typical commands are:

- Set timecode of IN point
- Set timecode OUT point
- Edit (record between IN and OUT points)
- Switch to Δt mode
- Record 1 frame in Δt mode
- Step forward 1 frame
- Inquire current timecode

9.5 Automating the Recording Process

Each reel of tape has a 'directory' file (`.SYD`) that governs allocation of video 'files' on the tape. This is just an ASCII file listing the name and starting timecode of each scene.

The placement of picture files within a scene is determined by an 'exposure sheet' (`.EXX`) file. Each line looks like

```
JXA 100 200  2 4
```

This generates the sequence of filenemes, JXA100, JXA102, JXA104,... and says to repeat each frame 4 times. The file-number increment can be negative, allowing frame number sequences to go backward.

Video recording is controlled by a program called `SONYR`. This program reads in the appropriate `.SYD` and `.EXX` files and then loops through the exposure sheet, reading pictures from disk and sending the appropriate control commands to record them. You can tell it to record any frame number range within the scene, since, with insert editing, you can record frames in any order. Records via insert edit if a picture holds for > 7 frames, via Δt if it holds for ≤ 7 frames. An attempt was to make it hard (but not impossible) to overwrite something accidentlayy.

SONYR can write out a new version of the .EXX and .SYD files containing extra information about cumulative times and cumulative frame counts. This information is purely for human consumption.

There are 3 ways to identify each frame on the tape.

1. timecode
2. sequential frame number from the beginning of the scene

3. file name

SONYR has commands to translate between any two of these, given the information in the .SYD and .EXX files. Since these files are on VAX disk, you can run SONYR without the having the videotape mounted, or even turned on, and just use it as a database retrieval program. This is important. Some video systems record this info on the video tape itself, in vertical interval. You have to mount the tape to get at it.

Chapter 10

Production Logistics

10.1 The production cycle

10.1.1 Make-files and exposure sheets

Many frames in these types of animations are "freeze". To avoid recalcualting constant frames, there is a post processor program that reads the .MOV file and determines the frame number ranges where things change.

The output of this program is 2 files. One containing a list of MAKE commands for just those frames that change, and one which is a .EXX exposure sheet file for sending to the SONYR recording program. The latter is often edited or adjusted before usage, but the hard part, the frame counting, is done by the program. For example, if the keyframes were

```
KEYF A,100,0, 200,1
KEYF B,300,0, 400,1
```

the program would generate the MAKE file

```
MAKE xx,100,200,1
MAKE xx,301,400,1
```

This is another advantage of separation of functionality. This program just looks at the keyframe tables and sees where symbolic variables are changing. It doesn't know or care how they are being interpreted.

This is another advantage of frame nubmers starting at 1000. If some symbol and keyframe values are being used as a function definition, its keyframes (i.e. parameter) table entries are likely to be in the range -10...10 and so won't generate false changing frames.

10.1.2 Making the frames

Normally, the `ARTIC` program is just started up, a scene description file `xx.ART` is read in and the MAKE file is read in. Then you just stand back and watch the frames come out. On a few scenes, picture files require post-processing to merge with other images, but this is rare.

10.1.3 Disk space monitoring

The big game in running production is keeping the disk free. When you start calculation of a scene, you must figure out a projection of when it will finish and when the disk will next fill up with pictures. You must always keep in mind when the next event will occur requring operator intervention.

10.1.4 Allocation of space on videotape

Tapes are half hour long. It turned out to be easier to manage tapes if the animation for each TV program was on a separate tape. The time code on a tape used the 'hours' field to indicate the reel number, e.g., reel 5 will contain time codes 5:00:00:00 through 5:30:00:00. The timecode definition has extra, unused bits called 'user bits' that are sometimes used for this purpose, but many post production houses don't have a way to read them. The 'hours' field works fine as a reel number.

Space on tape is allocated and kept track of in the .SYD file on the VAX. This is just a text file that assigns a name and time code value for each videotape "file" or scene. It's simply a text file and can be edited with the text editor. That's the only way to delete a scene. You can only delete last scene on a tape, but sometimes need to if it was created by accident. You can also (carefully) change starting point of a scene to eat into the inter-scene gap if necessary.

Scenes are allocated to start on even 10 second timecodes. Leave 10 secs blank between scenes. This makes it easier to find scenes when reeling through a tape in fast forward mode.

Each scene has a 10 second freeze on the first frame and a 10 second freeze on the last frame. These are called 'handles'. They make it easier for postproduction editing to do fade in/out of the scene. If the beginning of a scene was a cycle the 'handle' was recorded as a 10 second repeat of the cycle.

10.1.5 Checking tape

Sony sometimes glitches on record in Δt mode. Must carefully check and watch for these glitches. When found, re-record the bad frames and the tape is usually ok. Some commands were added to SONYR to ease this process. When you see a glitch, immediately hit the STOP button on tape. Type the FINT command to SONYR. It reads current timecode from the tape machine, looks it up in its tables, and tells which frame number you are at. Then tell it to record frames form about 30 frames earlier to the current spot. (you can't expect to hit it exactly and re-recording extra frames doesn't hurt).

10.1.6 Deleting frames

Once frames are verified as being OK on the video tape, they can be deleted from the disk. We gave up on backing up picture files on digital tape since there were so many of them. Most scenes are so simple that it is quicker to re-calculate a picture, if needed, than to find it on tape anyway.

10.2 Parallel processing

Parallel processing in this context means not just using different machines, but running different jobs on the same machine. But `ARTIC` uses the Picture System, and rendering programs use the Frame Buffer. In order to be able to design and make test frames while another terminal is running a production job these need to be freed up.

Also, the production of frames is often I/O bound because of lots of .TMP files, so it's profitable to run several terminals at once. Saving a picture to disk sometimes takes as long as rendering it.

The capability of using several processes and machines shows a disadvantage of special purpose rendering hardware. You usually only have one special purpose hardware box, it's common to have multiple tasks or multiple CPU's.

10.2.1 Detaching Picture System

The easiset resource is to free is the Picture System. A version of `ARTIC`, called `DARTIC`, was simply linked with a dummy version of the PS library. This is the version used for production.

However, a lot of special versions of `ARTIC` have been generated. Having two versions of each of these gets clumsy. For this reason, a command was added to `ARTIC` to cause it to detach from the PS after it has started up. All further PS calls within the program are then ignored. It also looks to see if a PS exists on the current machine; if it's run on a machine not having a PS, it disables the calls right away. The general operation is then to read in a scene description, look at PS to see if it's OK, then issue the Detatch command and read the MAKE file. Can also use this for Zen design.

10.2.2 Virtual Frame buffer

A Virtual Frame Buffer means using a file to simulate FB memory. This is built into the low level FB access routines. When a rendering program opens the FB device, the initialization routine looks for a VMS global symbol called VFB. If undefined, it uses the real FB. If defined, it uses that string as a file name to use for the virtual frame buffer. It expects the file to already exist and not be in use. Currently manual allocation of which VFB files are in use on what terrminals. A virtual FB file is just a normal picture file saved in in dump mode. Note the need to keep FB contents intact between rendering programs to make overlaying of images work. The low level FB routine maps file into virtual address space and accesses it. When closed, file is still there.

10.2.3 Dividing Up the Work

How to allocate frame number ranges to calculate.

Sequential

Simplest, and the technique normally used.

Even/odd

If not sure we have enough time for single framing do even numbered frames first. Then, if time, do odd numbered frames.

Binary search

For testing, do every 32, then every 16 between, then every 8, etc. Not used a lot because its confusing.

Whatevers designed first

Might not have the whole scene designed yet, but can start calculating what is designed. This might not be the beginning of a scene. Lots of opportunities to screw up here.

Both ends to middle

When multiple terminals or multiple CPU's are available the operator must allocate scenes between them. Most independant scenes can be done in parallel with no further worries. If two are working on the *same* scene, the operator must allocate frame number ranges to the different tasks/cpus.

The best way to do this is to start one doing frames forward from begin to end. Start other one doing frames backwards from end to beginning. When they cross, you're done. Usually can't predict accurately where they will cross so have to watch where they are. When they cross, manually kill the two tasks. Usually you have half saved picture file somewhere that must be found hard way.

10.2.4 Life in multiple processing mode

If I'm being *real* productive I can be on 5 or 6 terminals at once, simultaneously: Designing a new scene, Recording a finished scene on videotape, Calculating new frams on several terminals/CPU's, Deleting old recorded frames.

I have used up to 4 computers at once doing calculation. Image storage, recording and deleting all happens on the one VAX that has the Frame Buffer and videotape machine attached to it. DECNET allows remote machines to save their picture files directly to this one common disk. Can then check immediately if production run on remote node is doing right thing, very valuable.

To calculate on other CPUs you must copy all data files to the other machine. This may be tricky if some files are in libraries. A .SKL file can refer to files in one library. A Renderer can use several different libraries. You can wind up using old versions if you're not careful. This is a medium sized headache, keeping databases on 2 machines in step. Could have a common file system using DECNET, but it's slower and won't allow multiple jobs on the same scene because of temp file naming. In practice it didn't seem that hard making copies.

Many mechanisms are not bullet proff. Good enough for small scale shop where only a few jobs running at once. Making bulletproof is real hard.

When operating in this mode the room has many terminals with little signs on them telling: what scene they are running, what VFB they are using, which disk they are saving files to, what node of DECNET they are logged in at.

10.3 Final Delivery

When a videotape is full, or all scenes of one program are finished, I sit and record a 'scratch track' of narration for scenes from script This gives the production peple an idea of timing. Then I transcribe timecodes for each animation to a copy of the script. Make a 3/4" copy with burned in timecode to send with script for rough editing. The phrase 'burned in timecode' refers to an option on the 1" playback that superimposes the timecode in a little window on the screen, using a character generator. This enables the editor to easily see which frame corresponds to which timecode.

Part V

UNDERVIEW

Chapter 11

UNDERVIEW

11.1 Multiple Program Structure

You cannot expect to develop one program that does everything. It is much better to have a collection of smaller programs, each of which is relevant to some small portion of the whole process. This allows the sections of the process to be thought about and debugged individually without needing to consider all aspects of the system at once.

11.2 Unifying Techniques

There are three main unifying elements in the system discussed here. One is the output device, the frame buffer. Another is the input mechanism, a common command language interpreter. Finally there is the idea of common database formats.

11.2.1 Frame Buffer Synergy

The fact that all image generation programs use the frame buffer as their output device produces what I call the "Frame Buffer Synergy". This occurs because the frame buffer enforces upon many programs a common database for representation of images, namely the pixel array. Different programs do different things to the frame buffer but the output of one program is automatically suitable as the input of another if they all operate on the frame buffer. A system, then, naturally accumulates a collection of different programs that "do things" to the frame buffer. They may be written at different times, by different people, without any knowledge or

expectation of each others' existence. But they all can be used to aid in the production of the same picture.

11.2.2 The Command Language Interpreter

The other major unifying mechanism is a common command language interpreter which most programs use to process their input. While the command syntax is very simple, having all programs use it allows for several programs that generate data in this format to be able to talk to each other.

11.2.3 Data Format Standardization

This is the most important unifying strategy of all. Almost all databases are stored as ASCII text commands in a syntax that looks like CLI commands. I avoided usage of binary data files unless absolutely necessary. This means that you can always use the text editor to manipulate data. Lots of utility programs are lying around too to, e.g., accept a simple file of MOVE and DRAW commands and do some transformation and write it back out.

The communication of data between programs can take place in many ways. Some methods are facilitated by certain operating systems. Unix provides the useful mechanism of pipes, Multics allows dynamic linking of libraries, etc. Our system performs this communication with the logical equivalent, temporary files, usually in the format of command lists. One program would then write out a command file in the appropriate format for another. This is slower than some other schemes but it has the advantage of leaving a record of the intermediate stages of picture generation for easy retry and debugging purposes.

11.3 Flexibility

Once we have accepted the necessity of a diverse collection of individual programs in the system, the main problem becomes one of making them work together properly. This is mainly a problem of getting the necessary data to the appropriate program at the right time. Keeping track of which programs know about what data and how it moves from program to program is, I feel, the most important thing to know about a system in order to understand it. It is more important, in fact, than knowing the details of what the programs do with the data. The details of this process are, in fact, the main subject of this paper.

To maintain flexibility, the concept of "loose coupling" is important. This means that the system does not have to be "totally integrated". Certain groups of programs may be able to interpret the same data base, others

just some subset of it, others none at all. Various similar things may be done in different ways in different contexts. This may be contrary to what one might think of as the "ideal system", that is where everything is rigidly consistent and everything fits into one set of rules. The problem is that this is very constraining. It assumes the initial design was able to forsee all possible future needs and provide for them. Such a system stultifies experimentation with new ideas. An analogy can be made with various other natural and artificial systems, such as computer operating systems, the telephone system or the human brain. In each case, the most useful and flexible system is a conglomeration of (often incompatible) bits and pieces developed at different times for different purposes. We cannot allow the system to become too anarchistic, however. In parallel to the addition of new ideas to the system, effort must be continually expended to more completely integrate existing elements. The determination of when to follow the existing rules and when to be radically different is not really quantifiable and is more a craft than a science.

11.4 Various tradeoffs

11.4.1 General vs Special case

It is useful to have several programs that do effectively the same thing, but in different ways. The different ways refer to the degree of generality/specificity of the program to the image being generated. On the one hand, one may employ a general purpose program that has very general but low level primitives. The generation of the desired image then requires a large amount of manual input of data and parameters. On the other hand, one may employ a special purpose program which "knows" how to make a class of images such as the one desired. In this case much of the data input and positioning will be done automatically by the program based on its knowledge of the "universe of discourse". The general, but labor intensive approach is applicable to situations where only a few images are required of something not previously drawn. The latter case is applicable to situations where many images are required on a restricted class of subjects.

For example, we have a general purpose articulated object animation system which will draw arbitrary articulated objects consisting of arbitrarily shaped elements. The user must define the shape and placement of each object explicitly. On the other hand, there is a very special purpose simulation system for scenes in the solar system. One need merely say something like "draw a picture of Uranus as seen from the moon Miranda on Aug 28 1987". The program will automatically calculate the positions of the ob-

jects, pick a good view, and call upon the appropriate databases to produce the picture. An animation system benefits from having both these types of tools.

11.4.2 Clean vs Dirty implementation

It would be nice to have everything all figured out so that any scene is an obvious special case of the general purpose system. This sort of works. But there are always cases where really ugly hacking and bashing is needed. You try to be as aesthetic as possible, but sooner or later its TIME TO MAKE PICTURES. Then the chewing gum and bailing wire come out. It is necessary to go back after the fact and merge the new ideas into the existing syste. I hope I have the chance to do this.

Compare with: human brain, telephone system, MS-DOS. All these are really ugly kluges but they work.

11.4.3 Anarchy vs. Beaurocracy

Small scale project, anything works. Larger projects require some beaurocracy. Everybody has their favorite organizing principle, e.g., structured programming, object oriented programming. Mine is filenaming, modular progs, standard databases.

11.4.4 Interpreters vs. Compiled Code

The main problem of Computer Graphics: Evaluate

$$f(a,b,c,d,e,f,g,h,i,j)$$

for lots of pixels. But sometimes some of these are constant over the whole screen. Interpreters allow you to modify evaluation easily, viz. Rob Cook and Ken Perlin shading systems.

11.4.5 Menus vs. Keyboard Commands

I use keyboard driven programs because I can type better than I can draw.

11.4.6 Editors vs. Filters

Editors are characterized by reading and writing the same file type. Filters convert one file type to another. Filters are inherantly information destroying devices, e.g. a 3D rendering program takes a 3D description of a scene

and makes a 2D picture of it. The 2D picture no longer has all the 3d info (like the back sides of things).

Text editors are some of the most sophisticated software on compouter systems . lots of energy has been expended in fiddling with user interface. Programs that are editors for other than text can learn many valuable lessons from text editors, e.g. must save data automatically on exit, try vigorously to prevent user from accidentally exiting program without saving data.

11.5 Mistakes

11.5.1 Obsolete scenes

Early scene with torque used pale yellow, I remade scene the correct color but recorded corrected scene at another place on videotape. Production people wanted to reprise scene and got the old version since it was the first one on videotape. Shows you should destroy old obsolete scenes to prevent accidental use.

11.5.2 Mistakes in animation

- The tau in one of the scenes of torque is backwards
- Direction of Cerenkov radiation might be wrong.
- Cannonball/ground. Embarassing mistake. There is a scene of a cannon sitting on an irregular ground plane, firing a ball in a parabolic trajectory. As it fires we pull back to see parabolss. The problem is that the cannon and trajectory just shrinks, the ground doesnt change. I looked at this and felt uneasy but didnt realize what was happeneing until it was toolate.

11.5.3 Funny stories

Cannon/redcoat scene, where does cannon ball go? Russia

11.5.4 Inconsistant colors

Tried to keep it consistent but entropy creeps in.

- Blue color of electron changes from first half to second half.

- Blue-green color of area. Have colors for posn, area, volume but werent used properly for e.g. Gaussian surface. Had to fade wrong green into area-green in one scene.

11.6 Dead ends

11.6.1 Automatic storyboard generator

A pprogam was written to automatically time a script, allocate frame numbers to phrases of narration, and print out a storyboard sheet with blank spaces for sketches and each frame labelled with the narration. After a while this wasnt used, the ARTIC program was used as a live storyboard editor.

11.6.2 Movie table editing commands

ADJK command to adjust keyframe with knobs. Found this was not as easy to use as spreadsheet mode. RECO command records position of pen vs. time. It fills up the keyframe table too fast.